ICL TECHNICAL JOURNAL

Volume 4 Issue 1
May 1984

ICL TECHNICAL JOURNAL

Contents

Volume 4 Issue 1

ICL TECHNICAL JOURNAL

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

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

1984 subscription rates: annual subscription £14.00 UK, £17.00 ($37.00) overseas, airmail supplement £7.00 ($15.00), single copy £8.50 ($20.00). Cheques should be made out to 'Peter Peregrinus Ltd.', and sent to Peter Peregrinus Ltd., Station House, Nightingale Road, Hitchin, Herts. SG5 1SA, England, Telephone: Hitchin 53331 (s.t.d. 0462 53331).

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

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

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

Editorial

It will be obvious that there has been some change in the design of the cover with this issue of the *ICL Technical Journal*: in 1983 the company adopted a new house style – logo, colours and lettering – and this is embodied in the new cover.

Given that this change would be made, and that the Journal had now been in existence long enough for some stock-taking to be meaningful, the Editorial Board discussed other changes that might be made, both in the physical form and the content, taking into account comments that had been received from readers. For the first, the physical form, we came to the conclusion that only minimal changes were either necessary or desirable. The present format had been chosen after much consideration of the practical possibilities as pleasant and easy to handle and to read, and easy to carry around. All the comments we have received indicate that it is well liked, and the great majority – admittedly from only a small fraction of the total readership of over 6000 – are explicitly against any change. An important consideration to us was that this format and general appearance have established a clear identity for the Journal, familiar in many parts of the world; and this is something we should not wish to disturb.

We have therefore made only some small changes; a new typeface and an increase in the page margins. We believe these will improve the appearance of the page and add to the ease of reading.

On the content, again it seems that the level and general style of the papers are welcomed, as is the fact that most of the issues have covered a range of topics. However, there is much to be said for concentrating on particular themes, and the May 1983 issue, largely devoted to communications, was very well received. We therefore plan to publish 'theme' issues more frequently in the future.

The Editorial Board always welcomes comments and suggestions, and I can promise that anything of this nature sent to me is considered by the Board.

J. Howlett
Editor

The ICL University Research Council

P.D. Hall
ICL University Research Council Chairman

Abstract

The paper is in two parts. The first is a brief description of an organisation recently set up by ICL to encourage and support collaborative research projects between universities and the company. The second lists the 20 projects currently supported, together with a short note on each giving the aims and the motivation.

1 The Council

The establishment of the ICL University Research Council in 1982 was a recognition – some might say a belated recognition – by the company of the importance of academic research to the company's future.

Based on research carried out in Europe and the USA in the 20 years or so following the Second World War, in which the role played by a number of university laboratories was of fundamental importance, the computer industry has grown spectacularly to become one of the largest industries worldwide and certainly of vital importance to the economic health of the UK. In 1982 ICL recognised that its own earlier close relationships with academic research teams had not developed and that research in British universities was no longer making a proper impact on its development plans and its products. As a matter of Board policy, therefore, it was decided in the spring of 1982 that the company's future needs would be met increasingly by arrangements with the academic world rather than by expansion of the in-house capability. Recognising the need to isolate a programme of research to be carried out by a university team from the short-term pressures on ICL management, and the need for non-ICL input to the determination of research needs and plans, the company set up a body which it named the Universith Research Council, with three outside members.

The Council's remit from ICL was to concentrate on long-term research, and specifically to leave those short-term R & D projects which were contracted to universities to be funded by the company's development divisions or business centres. Recognising that the funds available to the Council were small compared with those of the Information Engineering Committee of the Science and Engineering Research Council (SERC) and that it would be of little value to ICL

to use these funds as an insignificant top-up to the latter, it was clearly necessary to focus activity in areas of particular importance to the company and to seek out areas of research which were, from the company's viewpoint, under-funded by SERC.

To assist in this task the URC supports workshops to which are invited research workers outstanding in the field to be studied, together with ICL R & D personnel. The URC aims to run eight to ten such workshops a year; the subjects of those held to date are as follows, and give a good indication of the areas of research judged to be of importance to ICL.

- the PROLOG language and PERQ
- logic and functional programming – tackling the issues
- novel architectures
- distributed UNIX
- expert systems and applications of knowledge engineering
- databases
- man-machine interfaces
- specification techniques
- measuring software throughout the life cycle
- impact of information technology on ICL's business and ICL's customers (seminar)
- future languages for development of information systems
- communicating sequential processes (seminar)

Workshops are restricted to a minimum of 24 hours to always allow time for informal discussion and numbers are limited by invitation to about 40 participants. They have certainly helped the URC in its consideration of where to spend its funds and also provided an opportunity for their audiences to make their own views known on what ICL should be about.

Since the URC got underway Alvey, Esprit and the ECRC – the European Computer-Industry Research Centre set up in Munich by ICL, Siemens and CII – have come on the scene as very welcome additions to Europe's research programme in information technology. The URC is now looking to support research relevant to ICL which does not fall within the remit of these programmes or which for some reason is best handled, at least initially, outside them. In particular, and as an example, it is looking to support research projects which will help to determine what 'Decision Support Systems' will be doing, and for whom, a decade from now.

The Council is now supporting 20 projects in British universities and is continually discussing new projects; a member of the appropriate ICL R & D group is closely and personally associated with every project. The second part of this paper deals with these projects. It is intended that wherever appropriate, URC projects shall form the subjects of papers in the *ICL Technical Journal.*

These activities have been an undoubted success from ICL's standpoint; and the

Council believes that the academic world has gained from this closer involvement with the company's long-term projects.

2 Projects currently supported by the URC

2.1 Introduction

The Japanese have recognised that within 20 years society will be faced with a potentially disastrous demographic change: the proportion of the population which is productive and wealth-generating will decline sharply in relation to the proportion which is dependent and wealth-consuming. The challenge of this situation can be met humanely only by rapid developments in the intelligent-systems industry based on computers and robotics.

Thus, society will place increasing demands on the computer industry. Hardware costs are coming down rapidly enough to meet these demands, so the problems are falling increasingly within the software sector. The ratio of software to hardware investment is rising dramatically, a current estimate being a factor of 100 every 5 years. This rate of increase will only be halted by developments in hardware and software technology, in particular in the following fields.

- Formal specification languages and techniques: Much of the cost of a software system is incurred during the implementation and testing stages of development, in correcting mistakes made in the design stage. The effects of these mistakes tend to accumulate, to spread and to become increasingly costly during the lifetime of the system. Formal specification languages and techniques can ensure that problems that need to be tackled during the design stage are tackled then and are tackled satisfactorily.
- Novel languages: Much of the cost of a software system derives from the low-level, machine-oriented and mathematically impure nature of programming languages. Their low-level nature makes them time consuming and difficult to use and their mathematical impurity makes them error prone in use. Components written in such languages do not lend themselves to reuse in different circumstances, with the consequence that the computer industry is continually churning over small variations on old themes. Novel and mathematically sound languages of the logic and functional programming types are overcoming these difficulties, taking the programmer to a higher, more problem-oriented and more reliable level of working.
- Novel architectures: Sequential hardware engines are reaching the limits of their performance capability. Increased performance for some important classes of problem has been achieved with single instruction multiple data (SIMD) engines such as DAP, but increased performance in general will need multiple instruction multiple data (MIMD) engines. These can be used effectively only with languages of logic and functional programming types, which need engines of this type of architecture to reach and surpass the performance of conventional languages. Thus, the future lies in the marriage of novel engines and novel languages.

- System architectures: A number of dichotomies have been introduced into computer systems over the years: for example between program and data, between programming languages and database languages, between programming languages and job-control languages, between filestore and virtual store, between distributed and nondistributed processing, between application and operating-system software, between catalogue and data dictionary and between hardware/firmware and software. Some of these stemmed from the needs to meet the performance constraints of the 1960s and 1970s and some arose as a result of the order in which new ideas developed in the industry. However, they are now seen as outdated and seriously hampering the work of the application system designer and must be removed wherever possible.
- VLSI and CAD: VLSI is the technology which will make it possible to produce the novel engines of the future; it will also enable us to produce hardware tailored to particular application requirements, for example spread-sheet hardware, just as we now produce tailor-made software. Hardware is increasing in complexity to mirror the complexity of software, and the techniques of formal specification languages and of the languages discussed above can and should be applied to hardware design as well as to software.

Even if we had solved the problems of software development we should still be faced with those inherent in interfacing computer systems with human organisations and environments. We have to plan the evolution of these systems and the environments in which they are used so as to ensure their synergistic growth. This involves the following considerations:

- Organisation modelling: As the interaction between computer systems and human organisations and environments becomes more complex we shall need to put more effort into designing and modelling these systems and their interfaces. The techniques we already have can be adapted to the modelling of the human organisations and environments, and also to that of noncomputer systems such as flexible manufacturing production lines.
- Flexible manufacture of software and hardware: Flexible manufacturing techniques are already being introduced into the production processes of many artefacts, enabling production to be tailored to the requirements of individual customers. These techniques will be needed also for the artefacts of the computer industry, that is, hardware and software systems.
- Expert systems: Techniques are being developed for capturing the knowledge of experts in particular fields, making this available for general use. These are increasing the synergy between the computer system and the human environment.
- Decision support systems: These are aimed at providing intelligent-system support for key managerial and technical staff in industry and the professions. They integrate the computer system into these key positions in the human environment.
- Man-machine interface (MMI): The language level of the interface between the computer and the user is being steadily, if gradually, raised and oriented more towards the human environment and the problem to be solved.

Consideration of all these points has governed the selection of projects for URC support. The remainder of this paper is given to listing the projects now being supported, with brief descriptions intended to show the essence and the aims of each. They are presented in groups of broadly related projects, but as will be clear the groups are not mutually exclusive.

2.2 Current URC projects

2.2.1 Group 1: Formal specification languages and techniques

(i) System architecture specification
Professor C.A.R. Hoare, FRS, Programming Research Group,
University of Oxford

The project aims to apply specification techniques to the description and continuing design of the ICL VME operating system. The work is at two architectural levels. The higher level is concerned with features such as block structure, context mechanisms, processes, virtual machines and synchronisation and the lower level with the detailed design of some components such as record management software, ring architecture and part of the data dictionary. The languages used are Pascal+ (i.e. Pascal with capabilities for concurrent processing), CSP (communicating sequential processes), Occam and Z.

(ii) Functional and concurrent programming specification languages
Professor R.M. Burstall, Department of Computer Science,
University of Edinburgh

The functional languages HOPE and ML, and the support system for the concurrent process specification language CCS, are being implemented on ICL PERQ and 2900-series machines; and the project is assisting in the use of these languages in particular for protocol description.

(iii) Reliability modelling of large software systems
Dr. B. Littlewood, Mathematics Department,
City University

This is aimed at developing practical models of software reliability. Most current models are in fact poor predictors of future reliability and do not concern themselves with the real-life problems arising from a large customer base using a given software system in many different ways. The project will attempt to develop new theoretical models addressing these problems, which will be validated using data from the ICL VME operating system.

(iv) Theoretical and practical aspects of software technology
Professor C.B. Jones, Department of Computer Science,
University of Manchester

The project supports formally-based techniques for software engineering and

tools by means of which these techniques can be made available to software engineers. In particular, it relates to an existing SERC-funded project to consider the development of tools supporting the Vienna Development Method of formal specification and rigorous development of software; it also supports a number of visiting experts and tool providers.

(v) Application of specification languages to CAD
Dr. G.J. Milne, Computer Science Department,
University of Edinburgh

Some support has been given to Milne and Taub in their research into the silicon compilers of the future. As the size and complexity of VLSI chips increase it becomes more and more necessary to adopt formal design approaches involving formal specification and automatic design translation. A further problem of interest is the formal proof that the silicon compiler itself does not contain errors. Building on the CIRCAL concurrent specification language, the Edinburgh group have been investigating the use of LISP on PERQ to specify and verify the designs of small VLSI cells.

2.2.2 Group 2: Novel languages and architectures

(i) PROLOG-X system, support environments and implementations
W.F. Clocksin*, St. Cross College,
University of Oxford

PROLOG-X supports the version of the logic language PROLOG described in the textbook by Clocksin and Mellish. This is a high-performance implementation with hooks to allow interfacing to graphics, database systems and other procedures written in high-level languages. The PROLOG-X system includes a program development environment for PROLOG written in PROLOG; it has been made available on ICL 2900 series and will be made available on PERQ, and a version based on UNIX is under development. The aim of the project is to provide a base for development of significant applications written in PROLOG.

(ii) Abstract machine design for functional programming, and related topics
Dr. P. Henderson†, St. Cross College,
University of Oxford

The project is concerned with the LispKit system, the portable implementation of the functional language LispKit LISP, described in Henderson's textbook, Functional programming – application and implementation. It has been made available on ICL 2900, PERQ and DRS, on the ICL Personal Computer and on numerous small machines. It has been microcoded on PERQ and is being microcoded on 2900. LispKit provides an excellent subject for experiments in microcode, hardware and system support of functional programming, and a basis for

* now at the University of Cambridge
† now at the University of Stirling

experiments in the use of functional programming languages. A personal database system and protocol simulation tool have been written in LispKit.

(iii) Logic system implementation, and databases
Professor R.A. Kowalski and Dr. K.L. Clark, Department of Computing, Imperial College, University of London

The project includes a programming development environment for MicroPROLOG, a version of PROLOG developed at Imperial College and described in the textbook by Clark and McCabe. This has been made available on ICL DRS and will be made available on PERQ, 2900 and the ICL Personal Computer. The project is now concentrating on the problem of interfacing PROLOG to databases in general and to CAFS in particular.

(iv) Concurrent hardware support for functional and logic programming
Dr. J. Darlington, Department of Computing, Imperial College, University of London

The project is concerned with parallel systems. The ALICE machine, designed by Dr. Darlington's team, is a multiple instruction multiple data concurrent processor machine, intended to support functional and logic languages. An intermediate-level language CTL is intended as a standard target language for compilers, which can then be further compiled into machine code for ALICE or other languages. An interpreter for CTL has been transferred to ICL 2900.

(v) Persistent programming and database systems
Dr. M.P. Atkinson, Computer Science Department, University of Edinburgh
Dr. R. Morrison, Computational Science Department, University of St. Andrews

The object of this project is to remove the differences now found between languages for programming, database description, data manipulation and job control, respectively, and to present one coherent language interface to the application developer. A language, PS-Algol, has been developed which incorporates these ideas, the PS standing for 'persistent' to indicate that the language may be used to access the persistent data of a database as well as the transient data of a program workspace. PS-Algol is being made available on ICL 2900, and applications developed using the language will be made available later in the project.

(vi) BASIX – a language for programming distributed systems
Professor B. Randell, Computer Science Department, University of Newcastle upon Tyne

The project is concerned with the development of a new language, BASIX, and along with this programming models which take into account the movement of computing from concepts of centralisation and sequentiality to those of decentralisation and parallelism.

2.2.3 Group 3: Expert systems and decision support systems

(i) Expert systems and PROLOG
Professor R.A. Kowalski and Dr. K.L. Clark, Department of Computing, Imperial College, University of London

The Imperial College team has developed a number of expert-system tools based on MicroPROLOG. The project is investigating the application of these in real-life situations, including official regulations of the UK Department of Health and Social Security and of the British Nationality Act; cartographic and software engineering applications are also being studied.

(ii) Dictionary concepts using linguistic techniques; language design
Professor G.N. Leech, Department of Linguistics,
University of Lancaster

The project is supporting the development of a system, using grammatical tag analysis of English texts, for checking spelling and typing. The text-correction system is being designed to recognise errors not only through spelling but also through the detection of unlikely combinations of words. The project aims to improve the success rate of the tagging system beyond its present 96·5% and to lead to a system for analysing the grammar of any English text, and also to the development of a computer dictionary of English. The dictionary will store words together with information concerning their grammatical classification, the way they occur together, their idiomatic characteristics and their meaning.

(iii) Text composition on PERQ
Professor J. Smith, Department of Computer Science,
Queen's University of Belfast

The project is concerned with the representation, storage and retrieval of characters; the emphasis is strongly on accurate representation, so that the final reproduction is of very high quality.

(iv) Impact of computer-based information systems on high-level decisions
Dr. G.W. Winch, Business School,
University of Durham

The project attempts to shed light on those aspects of decision processes at senior-management level which may be supported by computer-based decision aids. The intention is to focus particularly on discretionary, interactive use of such aids by senior managers when involved in unstructured decision making concerning strategic or policy issues.

(v) Intelligent business systems
Professor P. Henderson, Department of Computing Science,
University of Stirling (in collaboration with the Departments of Management, Business and Accountancy)

The project will study the use of functional programming methods in various applications in the fields of small business and local government. One aim is to identify the improvements needed in both hardware and software support.

2.2.4 Group 4: man-machine interface (MMI)

(i) Man-machine interaction, computer graphics and multi-dimensional modelling
Dr. C.J. Garratt, Department of Chemistry, University of York

The aim is to produce graphical display software to aid modelling of molecules. Three-dimensional dynamic images are required, composed of intersecting spheres representing the electron distribution in the molecule; no existing display provides the required combination of speed and quality. It is believed that DAP will be an attractive vehicle for the implementation of the system.

(ii) Workstation design, raster graphics
Dr. I. Page, Department of Computer Science and Statistics,
Queen Mary College, University of London

The project is to investigate the hardware and microcode structure for a two-dimensional Raster-Op engine suitable for use in a bit-map display system; and the application of such an engine in a single-user workstation.

(iii) Speech compression
Dr. C. Xydeas, Department of Electronic and Electrical Engineering,
University of Loughborough

The aim is to solve some of the problems encountered when attempting to enable office systems to handle both text and spoken information with equal facility, as is required by the convergence of the telephone with the computer workstation. A number of speech-compression schemes are being developed and assessed to meet a range of requirements for medium/high compression, economic, real-time implementation and high quality.

Acknowledgment

I am much indebted to G.D. Pratten of ICL's Mainframe Development Division for the notes on the individual projects.

The Atlas 10 computer

T.L. Faulkner
ICL Atlas Division, West Gorton, Manchester
C.J. Pavelin
Science & Engineering Research Council, Rutherford Appleton Laboratory, Chilton, Oxfordshire

Abstract

The paper is concerned with a very powerful and technologically very advanced general-purpose computer, compatible with the IBM S/370 principles of operation. The machine was designed and is manufactured in Japan by the Fujitsu company. It is the highest of a range of IBM-compatible machines and is marketed and supported in the United Kingdom and other countries by ICL under the name of Atlas 10, as part of an agreement between the two companies. The first part of the paper, by T.L. Faulkner, deals with the architecture and the more notable technological features of the machine; the second, by C.J. Pavelin, gives a user view from the Rutherford Appleton Laboratory of the UK Science & Engineering Research Council, where the first Atlas 10 in Britain was installed in April 1983.

Part I: Architecture and technology of the Atlas 10 computer

1 Background

As part of ICL strategy for intercepting technology in the design and development of the 2900 range of computer mainframes and systems, discussions were held with the Fujitsu company of Japan in 1981. These led to agreements in October 1981 to co-operate in the field of very advanced technology and for ICL to market and support, in the UK and certain other territories, a large machine already designed by Fujitsu itself, the machine given the name Atlas 10 by ICL. In addition to the main processing part of the system ICL is marketing a full range of compatible peripheral and communications equipment.

The Atlas 10 has been designed to be fully compatible with the IBM System 370 principles of operation. Its architecture and its basic design include features which enable it to incorporate extensions to these principles, for example the recently announced IBM extended architecture (XA). The IBM-compatible market is a very large one and is of strategic importance throughout the world; the agreement with Fujitsu gives ICL access to that part of this market where the need is for the largest and most powerful systems. Atlas 10 is a single-CPU system which is more powerful than the 'dyadic' IBM 3081 Model K. A dual-

processor system can be assembled which, at the time of writing, can claim to be the most powerful general-purpose system available.

The name Atlas 10 was chosen for historical reasons. One of the classical computers was the Ferranti Atlas, which pioneered many of the features now taken for granted such as virtual storage, paging, interrupt control, microcode and the operating system. It was designed by a Manchester University team led by Professor Tom Kilburn FRS and manufactured initially by Ferranti; the Ferranti Computer Division later became part of ICT and thus of ICL when the company was formed in 1968. The first Atlas was installed in the Atlas Computer Laboratory at Chilton in Berkshire in April 1964, in the same building in which the first Atlas 10 was to be installed in April 1983. The installation and commissioning of the two machines is itself an interesting comment on the progress of computer engineering and technology over the past 20 years. The Atlas, roughly a ½ MIP (million instructions per second) machine with a ferrite-core main store equivalent to 288 kbytes, was transported from Manchester to Chilton in 19 truckloads, took a month to instal and a further four months to commission for service. Atlas 10, a 15 MIP machine with a semiconductor main store of 16 Mbytes, was flown from Tokyo to London on a Sunday night, taken from Heathrow to Chilton in a single truck on the following Tuesday, installed Wednesday and was a fully operational machine on the Friday – Friday 29th April 1983, to be precise.

2 The IBM architecture

Some features of this are discussed in Dr. Pavelin's paper; the following is more of an overview. The significant fact from the point of view of a manufacturer is that it is a very firmly established architecture on which not only IBM products are based but also those of many other manufacturers, the so-called plug-compatible manufacturers (PCMs). The total investment worldwide in hardware and software which is tied to this architecture is very large indeed, and therefore the market it represents is, as has been said, of great size and strategic importance.

The architecture, known as the IBM System 370 principles of operation, has now had a life of 20 years; while there have been many small changes and enhancements over that period, there have been two particularly significant changes, at intervals of roughly 10 years. The facts of commercial and industrial life, for both manufacturer and user, dictate that with so large an installed base this is about the maximum frequency with which large changes can be introduced.

The architecture first appeared in 1964 with the IBM 360 series, from which the 370, 303S, 43XX and 308X series have evolved by a continuous process. The System 360 architecture was characterised by the 8-bit byte, storage addressing of up to 24 bits, an instruction set based on 32-bit words, common peripheral interfaces and a batch operating system – OS/360, later simple OS – which was to be used on any machine designed to the specification of the architectural

interface. The aim of this last was, of course, to make it possible for a user to replace a processor by a more powerful member of the series without losing any of the investment in software.

The early- to mid-1970s saw the first major shift in the architecture with the introduction of the System 370 machines. The concepts of virtual store and virtual machine were now included, with operating software developments moving forward in three main directions. The original OS was developed to use the virtual-store implementation, with dynamic address translation later performed by hardware, and has become a large system which itself needs a large system configuration to perform effectively. A smaller disk-operating system, originally called DOS, has evolved for use mainly on the smaller processors, and a supervisory type of system control program, VM/370, has emerged for use on all machines. This has two distinct uses: first, it can act as a 'host' to one or more 'guest' operating systems, and hence allow one mainframe to support concurrently several different types of operating software. It can be used, for example, to continue production work with an older system while developing a new one. Secondly, VM can be used as an operating system itself without any 'guests'; this is usually done with CMS, a conversational monitor system, to provide a popular transaction processing environment.

The mid-1980s are now witnessing the second major shift in the architecture, aimed at meeting the needs of today's largest users, whose systems are constrained by some of the original limits set in the S/360 and S/370 design. This 'extended architecture', or XA, addresses the three main problem areas in the existing system architecture; operating system overheads, real- and virtual-store addressing and, thirdly, input/output bottlenecks on systems with high throughputs.

The original design of what is now the largest operating system, MVS, envisaged separate programs running concurrently in separate virtual-address spaces, with limited communication between them controlled by access to common software. The need in practice to communicate more fully, and to share programs and data between address spaces, is now realised and a 'dual address space' concept has been introduced which, by using a system of primary and secondary addressing, allows direct communication between address spaces and thus avoids the overhead of waiting until the common software procedure is free for use. The storage constraint arises from the original 24-bit addressing; this limited both real and virtual store sizes to 16 Mbytes, a serious handicap in today's needs. XA now provides 31-bit addressing, thus theoretically allowing up to 2 Gbytes both for real store and for each virtual address space. On the I/O handling side, the original limit that only 16 channels could be addressed directly from either CPU is removed by the use of subchannels up to a theoretical limit of 64 thousand, all directly addressable from any CPU, multi-CPU configurations now being allowed. I/O bottlenecks in the operating software are avoided by building intelligence into the I/O hardware to determine the best routing for commands and interrupts.

At both major shifts in the architecture the aim was to protect the investment in

users' current software by providing modes of operation in which programs obeying the previous rules can still run. The architectural extensions allow the new concepts and methods to be used in new programs being developed for new and extended applications.

3 Atlas 10

3.1 Architectural principles

It will be seen from the preceding discussion that the architecture to which the Atlas 10 is designed is not static but is in a process of evolution. The driving forces for change are the need to overcome the current weaknesses in the design as they become apparent to leading users, and to take advantage of various technological developments as they become available, in different parts of the system. The pace of change is, however, very much regulated by the large size of the user base and the need to protect to a high degree the users' investment in current hardware, software and the management of their business. Therefore, although the architecture is evolving, it is possible to forecast future changes and thus to prepare for them in time to meet the future needs of the great majority of the market. This is the principle on which the design of the Atlas 10 has been based: the system meets all the detailed requirements of the System 370 architectural specification (and therefore supports all the IBM S/370 operating software) and the overall design anticipates the changes coming with XA. A balance of hardware, microcode and firmware implementations provides sufficient flexibility to allow the detailed requirements of XA to be incorporated into the later production models without unwarranted impacts or performance penalties. The design also allows for machines already supplied at the previous architectural level to be enhanced in the field to the level at which XA is fully supported.

Fig. 1 is a functional diagram of Atlas 10, showing how the system has been partitioned to give the best performance and flexibility while retaining compatibility with the S/370 architecture.

The hardware is arranged to follow the S/370 concepts of CPU, main storage, channels, device control units, devices and communications controllers, with the use of standard I/O interfaces and order codes. The system can be expanded with store modules, extra channels and a second processor. A key feature is the use of a global buffer store within the main store control unit; this provides a very large, fast access cache store accessible from both the CPU and the channel processor and gives very short access times for most operations.

The original design used 31-bit store address highways throughout, and a channel cross-call function which allows the CPU, or either CPU in a dual system, to address any of up to 96 channels. The CPU itself obeys a wider instruction set than that specified for the S/370 and uses microcode techniques to fetch and action individual instructions. Some functions, such as the implementation of

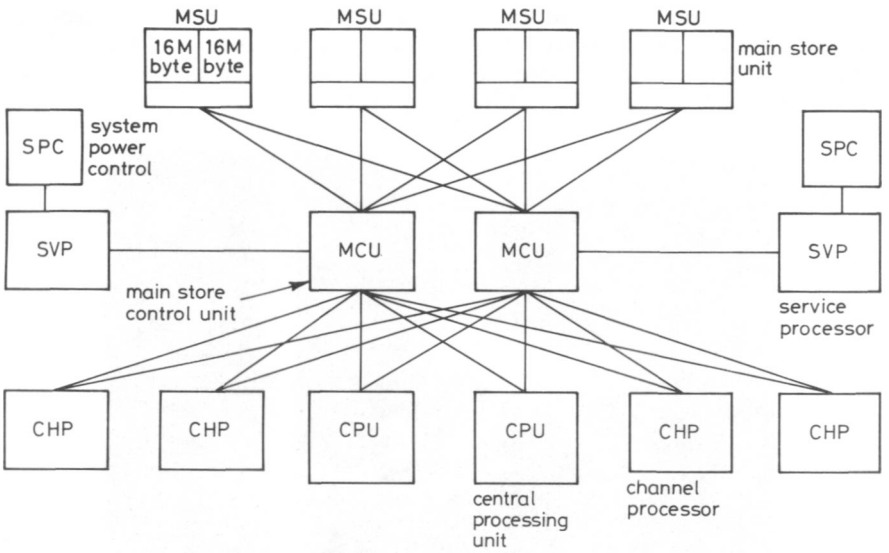


Fig. 1 Atlas 10 functional diagram (dual system)

dual address space and various 'assists' for running under VM, are also partly implemented by firmware; the CPU then obeys special machine-code instructions loaded into the main store at initial microprogram load. The use of microprogram and firmware techniques allows further enhancements to the architecture to be introduced with minimal hardware changes; and the use of the large high-speed caches in the CPU and MCU enables these to give a performance commensurate with the overall power of the machine. With dual address space already supported and 31-bit addressing included in the basic design, MVS/XA, the XA version of MVS, will be supported on Atlas 10 by the summer of 1984.

3.2 Engineering principles

The engineering strategy in Atlas 10 has been to provide the high performance by developing a technology capable of meeting the speed requirements of each part of the system. The use of individual components for each function is tailored to the performance requirements for that function, so that an overall economy is maintained for the design of the complete system.

For the central processing function a three-element major technological advance has been achieved in the development of LSI chips with their associated RAMS and their packaging on to PCBs and into modules.

The LSI chips, shown in Fig. 2, are designed in ECL technology with a basic master slice mounting up to 400 gates with a gate delay of 350 ps. For parts of the CPU logic the number of transistors per gate can be reduced, allowing 1300 gates to be mounted on a chip with the same gate delay. The chips are powered from −3·6 V with a maximum dissipation of 3 W. Heat transfer is by air cooling

from ten radiation fins mounted on a tower attached to the chip – these are shown in Fig. 2. Each LSI has 84 connection pins. The use of the 4 kbits RAM chip and LSI chips of this speed has made it possible to design the CPU with a clock cycle time of only 15 ns, the fastest of its type.

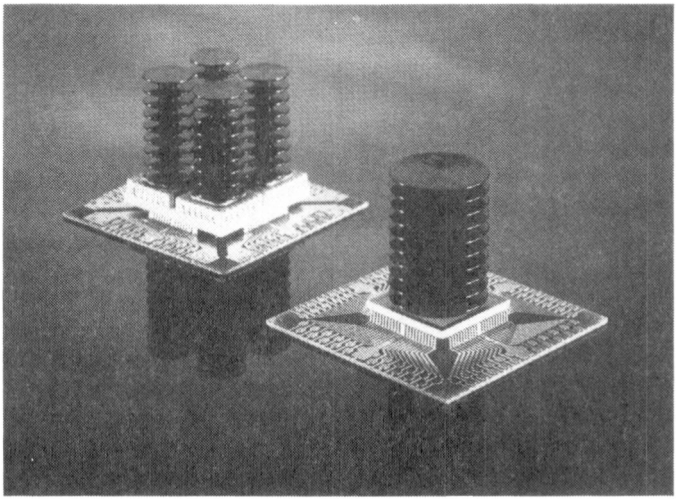


Fig. 2 RAM (left) and logic LSIs

Two types of high-speed RAMS are used with the LSI chips, of 4 kbits and 16 kbits. These are constructed with four chips mounted on a multilayer ceramic board, each having its own radiation tower with nine fins. The four-chip package has 60 connection pins. The 4 kbit chips are usually organised as 1 kbit x 4, with an access time of 5·5 ns, and are used in local buffer stores and control stores in the CPU. The 16 kbit chips, organised as 4 kbits x 4, have a 16 ns access time

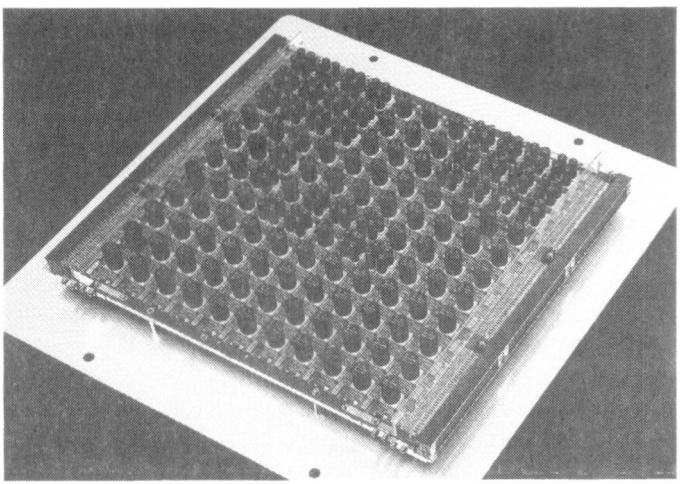


Fig. 3 An MCC, mounting up to 121 chips or RAMs

and are used in the global buffer store in the MCU to give the very fast access rate for a high proportion of store accesses.

The second element is the packaging of the LSIs and RAMs on to multiple chip carriers (MCCs), each carrying up to 121 packages; these are shown in Fig. 3. A multilayer MCC is about 1 foot square in size and shape, its 14 layers including eight layers for logical interconnection. Three off-carrier connectors, each with 192 pins, are mounted on each side of the MCC. To increase the tracking capacity and permit operation at LSI speeds without crosstalk Fujitsu have made the technological innovation of using tracks at angles of 30° and 60° in addition to the usual 90°.

The third element, illustrated in Fig. 4, is the housing of up to 13 MCCs in a 'cube'. The MCCs are fitted horizontally into what are called zero insertion force (ZIF) connectors mounted on multilayer panels at each side; logical interconnection is thus made by tracking in the side panels and not by means of the more usual coaxial cables which are slower, less reliable and more expensive. A special tool is used to move the MCCs into or out of the cube, which move the ZIF

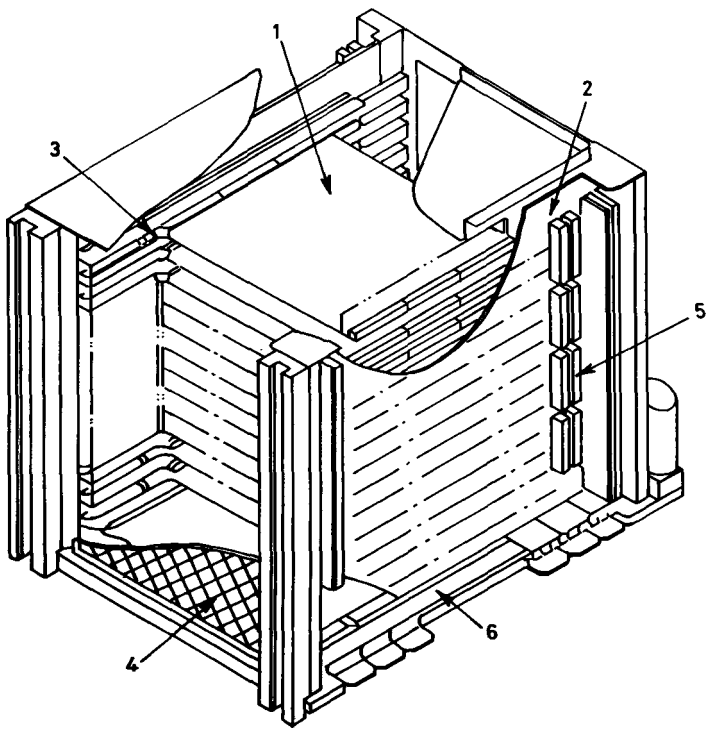


Fig. 4 Atlas 10 CPU: 'cube' construction

1 new MCC	4 air filter
2 side panel	5 I/O connector
3 Zif connector	6 power-supply plate

connectors from being in contact with both the side panel header pins and the MCC pins to a position where they are no longer connected to an MCC. This is shown in Fig. 5.

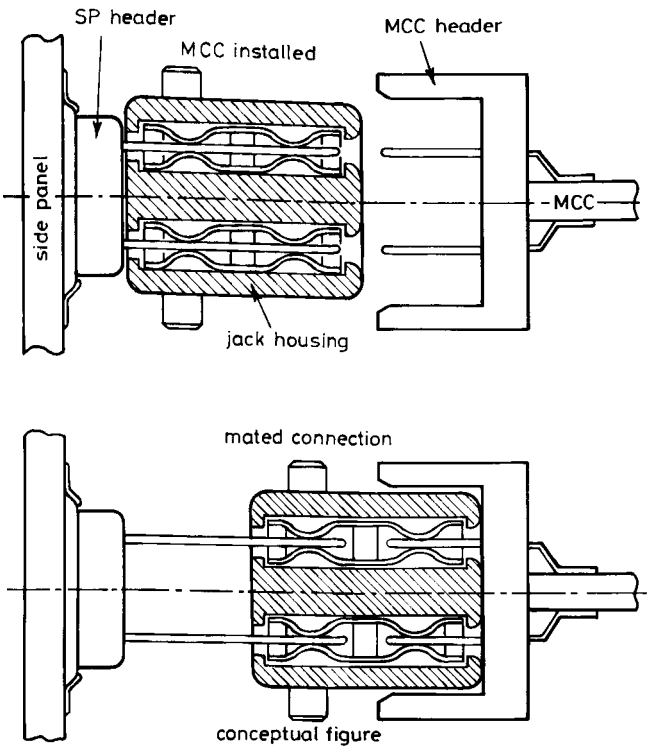


Fig. 5 Atlas 10 CPU: 'ZIF' connector

The complete CPU logic is contained on 12 MCCs, and so the whole processing power of this 15 MIP machine is contained in a single cube, the volume of which is not very different from 1 cubic foot. And despite the exceptionally high speed of the logic it is air cooled: chilled air is drawn from an underfloor void, taken horizontally over the MCCs of the CPU and of the MCU which is housed above it, and expelled from the top of the cabinet. Thus water cooling, with all its attendant problems, is completely avoided. Fig. 6 shows the cooling system.

The CPU and MCU are housed in one self-contained cabinet together with all the required AC/DC control and power supplies, and the main store and channel equipment are in a second cabinet. The two cabinets are linked with cable ductings. For the minimum single-CPU system, these two cabinets are required together with the console for the service processor (SVP).

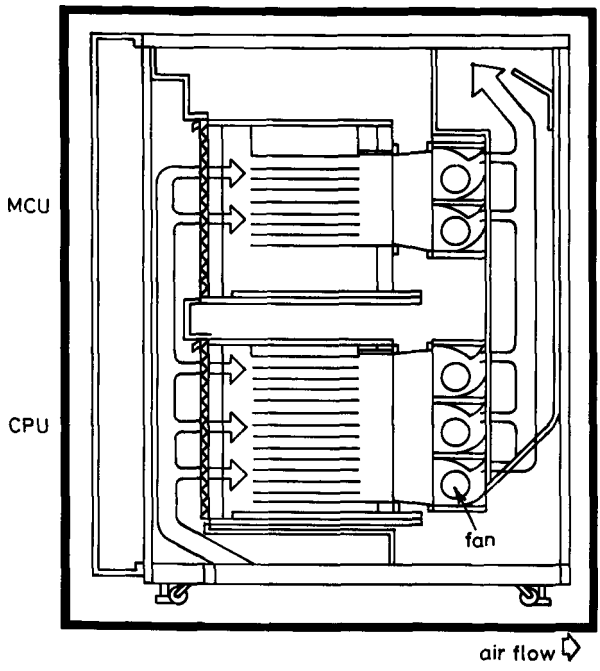


Fig. 6 Atlas 10 CPU: cooling arrangement

3.3 Design features

3.3.1 Processing: The CPU comprises three functional units: the local buffer store, the instruction unit and the execution unit. High-speed pipeline control is used, with independent pipelines and microprogram control for each of the instruction and execution units. The control stores use the 5·5 ns RAMs and are housed on the same multichip carrier as the logic which they control to give very fast operation. The CPU provides storage protection for blocks of 2 k and 4 kbytes, and for lower order addresses. High-speed floating-point, decimal and logical operations are also provided within the design.

A high-performance dynamic address translation unit is included. Using translation lookaside buffers each CPU in a system can modify the addresses for up to 512 pages by holding corresponding pairs of logical and real addresses which have been translated previously. A 'common segment function' allows the buffers to be shared between programs which use segments in common. Virtual-to-real address translation is further speeded up by means of a 128-entry stack in the CPU for segment table origins.

Execution of branch instructions accounts for a significantly high percentage of the total instruction time, sometimes exceeding 20%. Successes, meaning that the branch is followed, and failures occur almost evenly, and when a success

occurs in a pipelined CPU all instructions processed before the outcome is known are invalidated, with a consequent waste of time. Thus in the Fujitsu M-200, for example, most of the unsuccessful branch instructions can be processed in two machine cycles, whereas five cycles are needed when the branch succeeds. Atlas 10 has a special high-speed branch function to reduce this loss. When a branch instruction is detected in the instruction buffer this function calculates the branch target address before any execution of the instruction takes place, so that if the branch is unconditional the target instructions can be executed immediately after the branch instruction. If the branch is conditional both the instruction succeeding the branch instruction and the branch target instruction are fetched simultaneously and whichever is required is executed when the conditions are determined. The amount of time saved by this arrangement obviously depends on the actual software in use; but each successful branch saves two machine cycles.

3.3.2 Memory control and main store: Each main store unit (MSU) contains two 'segments' of sizes 8, 12 or 16 Mbytes: the term is not to be confused with the S/370 definition of a virtual store segment of 64 kbytes or 1 Mbyte. The logical data width of each segment is 64 bytes. To optimise the transfer to main store on large-store configurations the MSU is normally operated in four-way interleave mode so that the full power of the data path is used; otherwise the interleaving is two-way. The current design uses 64 kbit dynamic RAMs.

The main store control unit (MCU) contributes to the overall performance of the system by providing the large fast-access intermediate buffer store between main store and the CPU cache; also, it is linked to the service processor (SVP) through the system control interface and by means of this link provides the communication route for reporting errors to the service processor and controls the automatic system reconfiguration function. This last isolates a faulty CPU, main store unit or channel processor and reconfigures the system dynamically to use the units which are operating correctly; and can reincorporate a unit after it has been repaired, without interrupting operation. Dual systems with two CPUs can be partitioned into two single-CPU systems.

The storage protection keys are held in the system control interface; two keys indicated whether or not the protection block has been referred to and/or written to. To save time in accessing these reference and change bits a separate key buffer store for these is held in the MCU also.

Each main store control unit normally drives up to two main store units and up to two channel processors. A bus extension feature is available allowing both these numbers to be increased to four.

3.3.3 Channels: Each channel processor (CHP) handles up to 16 channels which can be of either byte-multiplex (MXC) or block-multiplex (BMC) type. The first are used by slow devices which transfer data a few bytes at a time and have a maximum rate of 80 or 110 kbyte/s; up to four of these can be attached to each

channel processor. The second are used by magnetic and other fast devices operating in burst mode. Two speeds are available: 2 and 3 Mbyte/s. Device addressing is organised to provide 2048 subchannels for each processor. A processor has a maximum throughput of 24 Mbyte/s, and so the maximum total throughput is 96 Mbyte/s.

A feature here is the inclusion in the design of dynamic address translation in the channel. Real store addresses can be derived by means of a virtual channel command word address which is issued by a special command which locks a channel program to operate in virtual mode; the address translation process is then followed in a similar way to that of the CPU. This has the advantages that an operating system designed to use this feature does not need to check page boundaries for chains and data areas, as is required with the S/370 indirect data address method, and free space arising from main store fragmentation can be used for more channel command words. Other special features are a command to lock a channel program into real address mode, and in dual configurations a channel cross-call to allow either CPU to address any device on any channel.

Each Atlas 10 channel is fully compatible with the S/370 channel interface specification, and special features are supported such as data streaming, which is required by the latest types of disk subsystem. Thus the mainframe can support compatible peripherals from other manufacturers, and equally Atlas peripherals can be attached to other compatible mainframes.

3.3.4 Service processor: This has three functions: it acts as the operating console of the system, it controls the configuration and reconfiguration of the system and it is used by the engineer for maintenance purposes. Some of the engineering functions have already been described; in its systems operating role it can perform the communications functions of the operating software – which may also be handled or supplemented at a terminal – and controls resets, initial microprogram load (IML) from data held on a floppy disk, initial program load (IPL) and dump functions.

A useful additional feature is the inclusion of a performance monitor, by means of which the service processor can display system operation states on a VDU, as follows:

CPU utilisation rates:	total utilisation rate
	supervisor-mode rate
channel utilisation rate:	utilisation rates of three selected channels

The display is in three colours and a light pen can be used for input.

3.4 Reliability provisions

Modern computer systems incorporate three tiers of design to provide high levels of reliability and availability: the intrinsic reliability of the components

and connections, the use of fault-tolerant techniques within the hardware and the co-operative use of recovery and fallback techniques by the operating software.

On Atlas 10, use of the advanced LSI, MCC and 'cube' technologies has reduced the number of interconnections by a factor of 10 compared with previous designs of CPU, and thus contributes powerfully to the intrinsic reliability.

The fault-tolerance techniques employed include the following:

- instruction retry at the CPU, in co-operation with the service processor
- Hamming codes for single-bit error correction and two-bit error detection in the microprogram stores in the CPU
- automatic fallback function in the CPU cache. If an error occurs the operation is repeated up to four times and if necessary a 4 kbyte section is deleted from the cache.
- duplicated, and in places triplicated, logic in the MCU
- extended error-correction code in the global buffer store of the MCU. This is a considerable extension to the Hamming code technique, enabling any error in a four-bit block of store to be corrected and any errors occurring within two such four-bit blocks to be detected.
- an alternate chip assignment function has been incorporated in the main store unit, in addition to the conventional Hamming correction and detection provisions. Independently of any CPU or channel processor accessing, the MCU periodically reads the contents of the main store unit. If a one-bit error is detected the data is corrected and written into the main store unit, thus correcting an intermittent error. At the same time the MCU checks all the bits in the 64 kbit RAM containing the error bit; if another error is detected the RAM is assumed to have a fixed one-bit error and is replaced with an alternative chip. By this means a two-bit error composed of fixed and intermittent one-bit errors can be prevented. One alternative chip is provided for each 16 Mbytes of main store.

4 Conclusion

The field experience from the installation and support of the Atlas 10 mainframe since the delivery of the first machine to SERC is witness to both the reliability provisions and the quality control processes followed during all stages of manufacture. Despite the size and power of the system, a reliability measured as several thousand hours between events is being obtained. The performance of the system has also been confirmed during benchmark experiments performed on behalf of customers and prospective customers, and by running benchmarks supplied by the independent Institute of Software Engineering. The results of these tests show that Atlas 10 is superior to all the competitive products, the advantage in scientific computing being very marked indeed.

Part II: A user's view of the Atlas 10

1 Computing environment

1.1 Introduction

Rutherford Appleton Laboratory (RAL) is the largest establishment of the Science & Engineering Research Council (SERC), the body which supports scientific research in universities and polytechnics in the UK. The Computing Division supplies large-scale computing facilities to SERC users throughout the UK and also supports SERC-promoted programmes in Computer Science research. The Division supports central (IBM compatible) mainframes for batch and interactive work, together with multiuser minicomputers and single-user computers (notably the ICL PERQ) located on site and in university departments. All these computers are networked on a private X25 network previously known as SERCnet, but now being taken over as part of the National Academic Network (JANET). The mainframes are linked to similar systems at high-energy physics laboratories in Geneva (CERN) and Germany (DESY).

1.2 Requirement for mainframes

Despite the large growth in the numbers of small time-sharing and single-user systems, a significant role is still seen for mainframe services in the SERC community throughout the rest of the decade.

One requirement for the mainframe is to supply the traditional 'batch processing' needed for analysis of data from high-energy physics experiments. A recent major international experiment at CERN involved 100 million proton/antiproton collisions ('events') of which five turned out to involve the W particle (the carrier of the weak nuclear force), which had been predicted but never observed. The event measurements were contained on 3000 (full) 6250 bpi magnetic tapes. Analysing the UK share of this data (about 300 tapes) needs over 1000 h processing time on a 5 MIP (million instructions per second) processor. This is typical of the combination of intense number crunching and data processing which is part of the contemporary 'big' experimental science. This type of application, together with numerical work arising from image processing, simulation, finite-element calculation etc. and the administrative needs of a large organisation such as SERC are seen as justifying 'traditional' mainframe batch facilities for probably the rest of this decade.

A large amount of interactive work – program development, office automation, database queries etc. – is also carried out on mainframe systems. Although at RAL the development of the mainframe interactive service arose as an adjunct to the batch, it is now regarded as very cost effective in its own right. Although it is

technically possible to run office automation or interactive databases over distributed systems, a shared mainframe makes the task very much easier once the scale of use can justify the cost. There will be a gradual movement of many of the functions of the mainframe interactive service out to single-user workstations, but for many years the mainframe will remain important in this role.

1.3 Pre-Atlas configuration

At the beginning of 1982 the basic processor configuration was as shown. Although the 360/195 'back-end' computers were about 10 years old, they each gave greater than 5 MIP performance on a scientific workload – they had in their day been among the fastest processors in the world. Since they were not virtual-memory systems they did not support MVS, the IBM mainline operating system for large processors. The laboratory was therefore running its predecessor, MVT, an operating system which had been unsupported by IBM for several years and which the laboratory itself had been modifying and enhancing for many years.

The 3032, which was used as 'front end', is roughly a 2½ MIP system which does support virtual memory. It was running IBM's 'hypervisor', a virtual-memory operating system in which each virtual machine presents an interface exactly like the real machine. Thus VM can run underneath itself multiple copies of IBM operating systems like MVS or MVT, or indeed VM itself. VM also supports the 'conversational monitor system' (CMS), a timesharing system specifically designed for VM. CMS had been chosen by the laboratory to support its mainframe interactive work, mainly because it was very cost-effective in terms of average users supported per unit of any resource compared with the IBM alternative TSO; this was so even although TSO runs as part of MVS (and MVT) and would have been more convenient. CMS gives fully interactive facilities and also an associated batch system. Fig. 1 shows a number of 'CMS virtual machines', one per user, plus support of the network, and the MVT system for communication with the back end.

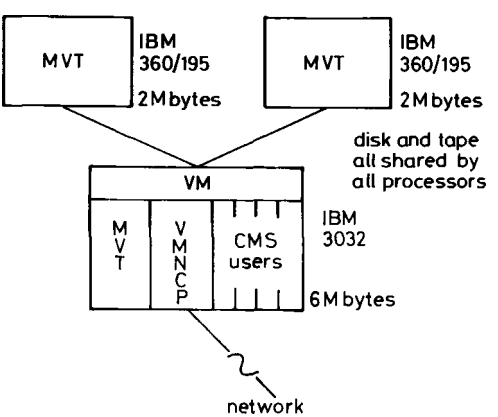


Fig. 1 Pre-Atlas processor configuration

A programme of replacement of the mainframe complex was becoming urgent for a number of reasons.

- The costs of maintenance and electricity associated with the 360/195s were very large compared with costs on contemporary equivalent machines.
- Although still surprisingly reliable, the 360/195s were at an age when substantial deterioration could be expected.
- MVT was an unsupported system – it demanded substantial effort to maintain and users were cut off from modern facilities.
- The resources were under strain, particularly the front end 3032 which had no cost-effective upgrade. In particular there was no spare resource even to begin the move from MVT to MVS, estimated at two years work to achieve in full.

1.4 The replacement programme

In November 1981 the laboratory invited tenders for IBM-compatible systems to replace the mainframe complex. In the same month ICL announced their agreement with Fujitsu allowing ICL to support and market the largest of the new Fujitsu IBM-compatible machines. Unfortunately ICL was not quite in a position to tender for machines (the formal launch of the product took place in May 1982). As a result of the tender exercise, the IBM 3081D was chosen to meet the front-end requirement and allow one 360/195 to be removed.

ICL then proposed the first Atlas 10, as the M380 had now been named, SERC was given approval to make a single tender invitation to ICL, provided that the technical criteria were met and that suitable contractual conditions could be negotiated. This proved to be the case, and the following timetable was followed:

July '82	delivery of 3081D, removal of one 360/195
August '82	3081 swapped to front end, 3032 to back
October '82	second 360/195 removed
April '83	Atlas 10 installed as second back end (MVT)
August '83	formal handover of Atlas 10
September '83	3032 removed from scientific use (for administration)
October '83	trial MVS service begins on Atlas 10

As this would be the first Atlas 10 to be installed in the UK, this extended timetable was planned to allow for possible difficulties. In the event, as the companion paper by Faulkner shows, everything went very smoothly indeed and the machine was fully operational only three days after its delivery to the site.

The final configuration is shown in Fig. 2. The whole change in mainframe configuration took place with no disturbance to users or to the general development programme.

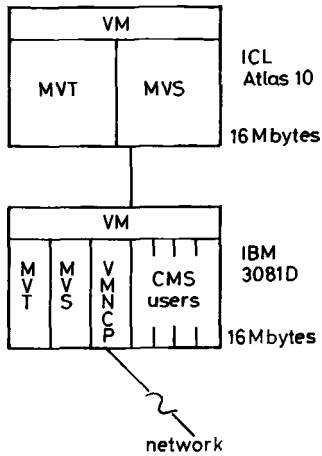


Fig. 2 Final Atlas configuration

2 Plug-compatible systems

2.1 Compatibility

RAL, like many other prospective purchasers of a 'plug-compatible' system, had to satisfy itself not simply on the usual range of issues – price/performance, reliability, support etc. – but on the key issue of compatibility with IBM. The term 'plug-compatible manufacturer' (PCM) applied first to those suppliers of equipment which attached to IBM channels or control units, and later to a new breed, led by Amdahl, who supplied processors compatible with IBM. Fujitsu, who had been manufacturing computers since 1954, introduced their first IBM-compatible machine (230 series) in 1974, and ICL entered the market in 1981. (Strictly speaking, it already had in 1965 with the System 4 range, but this was never sold as an IBM-compatible system, and it ran non-IBM operating systems. There are still System 4 computers supported by ICL in the field.)

When designing its mainframe replacement strategy RAL made a decision to remain with IBM-compatible mainframes and with the IBM systems VM/CMS and MVS. This was not simply because of the current workload and the problem of converting programs and data. IBM would continue to be a leading supplier of mainframe systems at the performance levels required by the laboratory, IBM and IBM-compatible systems were installed in collaborating laboratories such as CERN, DESY, Daresbury and others, and this would also continue. The IBM architecture is thus being chosen not necessarily because of intrinsic technical merit but for strategic reasons – the strategic importance of the IBM-compatible market is emphasised in the companion paper. The fact that there are competing suppliers for the hardware and for high-level software makes it possible for a public-sector organisation like SERC to take this view without favouring particular suppliers.

The Fujitsu computers are sold in Japan not simply as IBM-compatible systems. The assembler code interface is a superset of the IBM *'Principles of operation'*[1]. Fujitsu have implemented an operating system (OSIV F4) which is claimed to be an enhanced version of MVS with improved performance. There is similarly a Fujitsu equivalent to VM (known as AVM). In the RAL environment, although there may be much technical merit in alternative operating systems, the strategic effect of any departure from the 'standard', must be carefully considered. Thus RAL could not have incorporated the Atlas 10 unless it was clear that the standard IBM control software would run without any modification, and would be fully supported. At the time of the ICL/Fujitsu announcement, the attendant publicity stressed the alternative software, but it is now clear that ICL regards support of IBM standard software as paramount.

The version of MVS which was then the most recent was 'MVS system product, version 1, release 3', and it was crucial that this ran without problems. The order code had been enhanced in the latest *Principles of operation* to support MVS SP 1.3, and it was therefore not simply a matter of running new software on proven hardware – Fujitsu had to ensure that new hardware features were implemented. MVS SP 1.3 was not available in Japan when the RAL benchmark (see below) was first run, and so OSIV F4 was used with the proviso that the acceptance tests would involve repetition under MVS SP 1.3 (demonstrated in Japan in March 1983, before delivery).

IBM licence their control software for non-IBM processors, but (reasonably) do not undertake to look at any problems which cannot be duplicated on an IBM processor. Although RAL has a 3081, it is not always possible or convenient to duplicate problems even if there are genuine MVS bugs, and RAL will rely on the PCM supplier (ICL) to act as the prime MVS support on the Atlas 10. ICL will tell RAL, or even IBM directly, if faults turn out to be essential MVS ones. In fact the only specifically Atlas 10 MVS problem encountered was caused by an interaction with VM (see below).

RAL had a requirement for VM on the Atlas 10 for two reasons: to act as a back-up for the front end, and to allow MVS during the conversion phase to run in parallel with MVT. Ability to run VM SP release 2 was therefore also specified in the acceptance tests. Certain bugs were discovered; the most serious was a subtle problem in the area of maintenance of 'shadow tables'. These are real page tables which VM must maintain on behalf of operating systems running beneath it (a real copy of the virtual tables which the system believes it is operating on). An error caused MVS occasionally to 'hang' when the system was loaded. This was corrected by a change to the microcode.

2.2 Future compatibility

Apart from compatibility with current hardware, the other interest is the future. As the *Principles of operation* develop, how long will it be possible to upgrade the Atlas 10 incrementally to take advantage of the most recent software (and possibly peripherals)? The same question is of course asked when buying IBM

processors, but there is an assumption that IBM will have planned at least the more immediate developments and will have built the support for these in the processor. The microcoded nature of the Atlas 10 should make it fairly easy to respond to such changes. There is also a facility for interrupting into a hypervisor where ordinary 370 code can effectively act as microcode in implementing a new order. This means very rapid response to developments can be made, although at the expense of relative performance.
ICL has now stated that it will be able to upgrade the Atlas 10 to support the 'extended architecture' (XA), extensions to the IBM *Principles of operation*[2] which allow 31-bit addressing and enhancements to the channel architecture. Although RAL has no committed plan to move to XA yet, it is the obvious strategic path, and it is useful to have both main processors, 3081 and Atlas 10, able to support this move.

3 Performance

3.1 Processor power

The M380 processor was benchmarked simply by running a series of batch jobs for which timings were already available on the IBM 360/195 (MVT) and the IBM 3081D (MVS). The jobs were run both singly in series, and then altogether to confirm that a high level of multiprogramming had no untoward effect. Each job yielded a ratio M380:360/195 and Atlas: 3081, and the mean ratio (i.e. each job weighted equally) was

Atlas: 360/195	2·6
Atlas: one processor of 3081D	2·8

(The 3081 is actually a dual-processor system, so the throughput of the machine is potentially almost twice what the CPU times imply.)

The 360/195 was rated at about 6 MIP for this work (scientific), and the benchmark thus confirmed the 15 MIP rating. The individual jobs showed a wide variation: 1·3-5·3 in the case of the 360; 1·8-4·3 in the case of the 3081D.

This high single-processor performance has subsequently been confirmed in production scientific work. Some finite-element codes have given the following ratios of Atlas 10 to one processor of 3081D.

integer arithmetic	2·3
single precision matrix algebra	3·0
double precision matrix algebra	3·6

The Alas 10 is run transparently to the end users – when batch jobs are submitted they run on the 3081 or the Atlas 10 according to the loading of the system. The user thus does not directly see the power of the Atlas as the system is arranged so that on average there is an equitable charge. For jobs whose Atlas:

3081 CPU times vary from the average, the charges for CPU time are unfortunately different on the two machines.

The amounts of processing power, measured in MIPs installed, at various stages in the replacement programme are shown diagrammatically in Fig. 3.

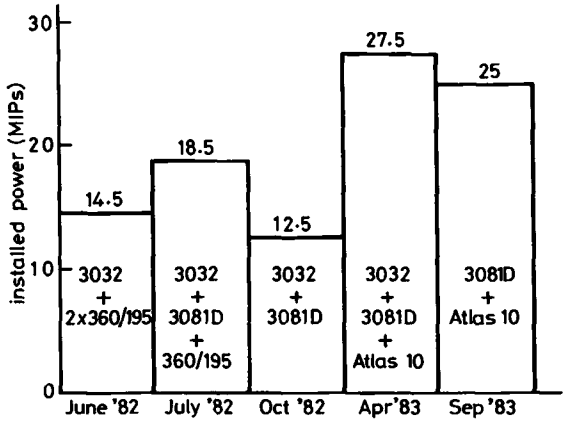


Fig. 3 Installed power for scientific computing during the replacement programme.

3.2 Channels

Between the Atlas 10 channels there are various subtle dependencies which must be taken into account when configuring data-streaming devices, i.e. those which run up to 3 Mbyte/s rather than the 1·5 Mbyte/s limit of the older style channel protocols. For example, the channels are in groups and within any one group there is a limit of 6 Mbyte/s on the total throughput. When first configuring the device layout, RAL was unaware of the impact of these dependencies and the initial positioning of the data-streaming devices caused performance problems which were difficult to diagnose.

3.3 VM performance

The Atlas 10 has been used in a front-end role, as this is necessary when the 3081 is down for any reason. It was noted that the processor times for certain operations under CMS were much higher than might be expected, and investigations showed this to be due to the implementation of 'VM assist'. This feature[3], is a facility to improve the performance of certain privileged instructions. A guest operating system issues privileged instructions in what it assumes is in privileged mode, but in fact, since it is running in VM, is in problem state. This causes an interrupt into the VM supervisor (CP) which emulates the action of the instruction and returns to the guest system. With VMA the interrupt to CP is avoided in certain common cases by an extension to the *Principles of operation*. On Atlas 10 VMA is implemented not in microcode but in a hypervisor which

runs in normal assembler code. Thus although the function of VMA is implemented, it does not have the effect of increasing performance to the extent expected, especially for programs running in basic control mode such as MVT. This is currently being investigated by ICL and Fujitsu.

4 Summary

To sum up, the experience of the first IBM-compatible system sold by ICL (System 4 excepted) and the first Fujitsu M 380 in the UK has been remarkably good. It is a tribute to the machine that the users were scarcely aware of the transition to the Atlas 10 – simply there was suddenly a lot more power available. There have been no 'compatibility' difficulties. The greatest problem arose from the assumption that the channels were mutually independent in performance; in fact there are idiosyncracies which it is essential to take into account when configurating the layout of devices on channels.

Acknowledgements

Thanks are due to the staff of ICL and of RAL Computing Division who co-operated so well in the installation of Atlas 10. Particular mention must be made of David Rigby of RAL Computing Division who unfailingly unearthed the most recondite technical detail.

References

1 *'IBM System/370 principles of operation'*. IBM Systems Library order number GA22-7000-7, IBM Corporation, Department D58, PO Box 390, Poughkeepsie, New York 12602, USA.

2 *'IBM System/370 extended architecture'*. IBM Systems Library order number SA22-7085-0, IBM Corporation, Department D58, PO Box 390, Poughkeepsie, New York 12602, USA.

3 *'Virtual-machine assist and shadow-table bypass'*. IBM Systems Library order number GA22-7074-0, IBM Corporation, Department D58, PO Box 390, Poughkeepsie, New York 12602, USA.

Towards better specifications

K.J. Turner
ICL Technical Directorate, Kidsgrove, Staffordshire

Abstract

This is a tutorial paper intended for the general reader who desires a broader knowledge of how to write precise specifications. To whet the reader's appetite for deeper study, various specification methods are illustrated using examples of familiar systems. The paper opens by defining what a specification is and how specifications can be compared. The problems of writing specifications in natural language are pointed out. A survey of specification methods is then given under three broad headings: informal, semiformal and formal. Finally, future trends in specification techniques and ICL's work in this field are explained.

1 Introduction

1.1 Specifications

What is a specification? The Collins dictionary definition[1], slightly edited is:

> 'A detailed description of the criteria for the constituents, construction, appearance, performance, workmanship, etc. of a material, apparatus, etc.'

This neatly puts over the point that a specification is about the *properties* of something, whether these properties be physical, logical, aesthetic, qualitative or whatever.

The word *specification* is usually used to mean a design specification. The word *implementation* is usually used to mean the realisation of a design. Specifications and implementations can thus be thought of as extreme ends of a spectrum. Here is an example of how a high-level specification from a customer might be elaborated through successive design stages to something a supplier might deliver:

Something to allow a loudspeaker to be sited remotely from its amplifier
↓
A 10 m loudspeaker extension cable
↓
(1) 10 m, two-core flexible cable suitable for low voltage, medium current.
(2) Cable to be terminated with standard two-pin DIN connectors
↓

(1) 10 m of cable type . . .
(2) Cable to be terminated with socket type . . . at one end and plug type . . . at other end in accordance with DIN standard . . .
(3) Cable with connectors to be supplied in package type . . . with legend . . . at price . . .

At each of these steps the specification is made less abstract and more implementation-oriented, until finally something which can be delivered is reached. The steps from a specification to an implementation constitute the process of *refinement.*

1.2 Comparison of specifications

In what sense can one specification be said to be better than another? Obviously a specification has a number of desirable attributes, such as being:

- easy to read
- clear
- free from errors
- precise
- appropriate
- complete.

What is not as obvious is how these attributes can be captured in an objective way so as to permit comparison of specifications. The work of Gilb[2] has shown that it is possible to describe attributes in a meaningful and measurable way. Some attributes translate easily into objective measures: for example, part of being 'easy to read' is that a suitable range of typefaces and typesizes is used. Other more subjective attributes such as being 'clear' could be measured by asking readers to complete a comprehension test after reading a specification.

In general, then, it is possible to quantify how good a specification is and therefore to show that one specification is better than another for a given set of measurable attributes. For at least two reasons it is desirable to aim for high-quality specifications. First, specification writing is a major part of the work of any business; ICL's Product Specification Document library alone runs to about 1900 documents, totalling tens of thousands of sheets of paper. Secondly, the specification of requirements and design is the start of a long and costly development process. Any errors or misconceptions introduced at the specification stage are likely to be repeated and magnified in later stages. It is typically estimated[3] that it costs 400 to 500 times as much to fix a fault once a computer product has been installed compared with correcting the errors at the requirements stage.

1.3 Using natural language

Most specifications are written in natural language (English, French etc.). The vagaries of natural language have been very amusingly illustrated by Hill[4]; most of the following examples are taken from his paper.

Ambiguity
police club visitors (on a sign)

Double negatives
I don't know nothing

Scope
'I feel like going to bed with Brigitte Bardot again tonight.'
'Again?'
'Yes, I have felt like it before.'

Program-like constructs
Shampoo, rinse and repeat (indefinitely?)

Punctuation
'BR hope to have trains running normally, late this afternoon',
printed as 'BR hope to have trains running normally late, this afternoon'

1.4 Purpose of this paper

The moral is clear, that using natural language in a precise way is difficult. Legal language, of course, aspires to this, but is notoriously difficult to read. So what is the way forward? A variety of techniques are now available which make better specifications possible. The purpose of this paper is to bring descriptions of some of these methods together in one place. Section 2 provides a broad classification of specification methods into informal, semiformal and formal techniques. Sections 3, 4, and 5 expand on each of these and show some of the methods at work on small problems. Section 6 rounds off the paper and explains what ICL is doing to improve its own specifications.

Because this is intended to be a tutorial paper for the general reader, only a small part of each specification method is described and illustrated. It would therefore be wrong to conclude from the paper that the methods are trivial and offer no advantages over traditional methods. It would also be wrong to conclude that the methods are applicable only to trivial problems. No attempt has been made in this paper to compare and contrast the methods since this would have required a more detailed exposition. However, the reader is encouraged to try the examples so as to get a feel for the methods. The References section gives guidance on further reading if more information is required.

The choice of which methods to mention and which to illustrate was a difficult one for the author. There are probably hundreds of specification methods, counting their variants. The criteria for selecting the methods cited in this paper were that they should be important, used in ICL and suitable for illustrative purposes. The interested reader will find a great wealth of papers and books covering methods not mentioned.

2 Specification methods

2.1 Broad classification

By way of a broad classification of specification methods, the terms *informal,*

semiformal and *formal* will be used. There is no clear definition of what formality means[5], but a common view is that a formal language consists of symbols plus rules for combining and manipulating symbolic expressions.

At one end of the spectrum informal methods are those without any (overt) mathematical basis. Informal specification languages, such as ordinary English, abound. They are a convenient and natural way of expressing a great deal more than specifications, but their lack of precision and mathematical basis makes them unsatisfactory where accurate descriptions are needed.

It is convenient to introduce semiformal methods as an intermediate classification. These methods can be given some sort of mathematical rigour, but are generally not used in this way. The emphasis is less on proofs of system properties and more on conveying ideas in a readily understood form.

At the other end of the spectrum lie formal methods, which fall into the province of mathematics. The familiar rules of arithmetic, for example, constitute a formal system. Do not be put off by this emphasis on mathematics. All of us do elementary arithmetic, for example, without having to know the intricacies of number theory. Similarly, it should be possible to use specification languages without being highly proficient in mathematics. The great advantages of using formal methods are that precision is assured and that formal proofs can be conducted. Just as at school we learn to prove theorems in geometry using standard rules, so a computer program can be mathematically checked against its specification to see if it has correctly implemented it.

2.2 Detailed Classification

Fig. 1 attempts to categorise specification methods under a number of general headings. It should be stressed that this is a personal classification and is not

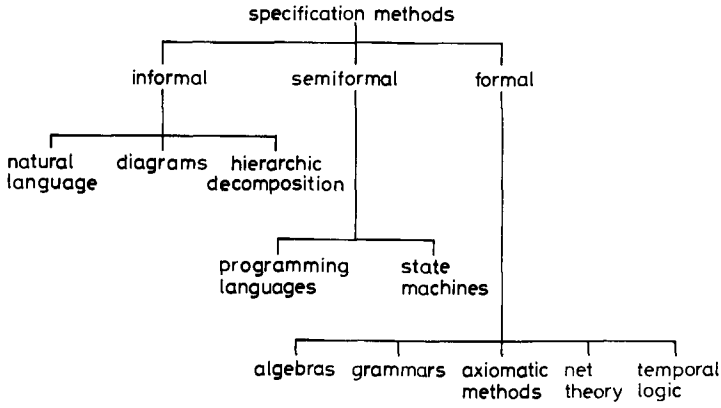


Fig. 1 Classification of specification methods

exhaustive. In particular, the three classes of method are only broadly distinguished and it could be argued that some methods should appear in an adjacent class. Based on the categorisation given, Sections 3, 4 and 5 now discuss examples under each of these headings.

3 Informal methods

3.1 Natural language

The use of natural language to write specifications has already been covered in Section 1.3.

3.2 Diagrams

Diagrams are often used to complement textual specifications. Many kinds of stylised graphical representation are used. Flowcharts[6] for computer program design are well known, and exist in many varied forms. Less well known, perhaps, are the techniques for modelling entity and function relationships. As an example of this genre, part of the D2S2[7] (Development of Data Sharing Systems) methodology will be described.

D2S2 can be used to model the relationship between *entities*, which are the objects of interest; they could be files, events, processors, systems or whatever. Entities are shown as boxes with rounded corners. Between entities there exist relationships. Lines of various types are drawn between entities to show different relationships (e.g. optional, mandatory, one-to-many). Fig. 2 shows an example of this technique to describe the *ICL Technical Journal* itself.

3.3 Hierarchic decomposition

Hierarchic decomposition techniques work by forcing the specification to be given at a high level first and then broken down into successively smaller parts. This is, of course, a commonplace design procedure: the hierarchic decomposition methods merely make it systematic. There are so many methods of this type that it seems almost invidious to single any out for mention. However, HIPO[8] (Hierarchical Input-Process-Output) is cited because of its widespread use by IBM.

The specific method to be illustrated, however, is DBO[2] (Design By Objectives). This is chosen as an example because it is different in character to most of the specification methods mentioned in this paper. DBO is ideally suited to writing requirements specifications rather than design specifications. There are three basic principles behind DBO. First, complex things are decomposed progressively into simpler ones. Secondly, the *functions* ('what'), *attributes* ('how well') and *solutions* ('how') of a problem are carefully distinguished. Thirdly, all attributes are quantified in a meaningful and measurable way.

Suppose, for example, that one treated having a meal at home as an activity to be

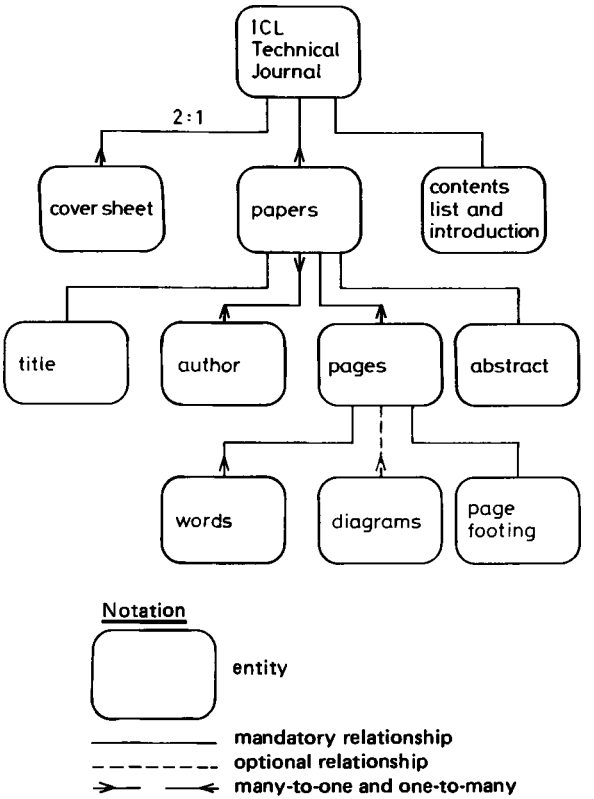


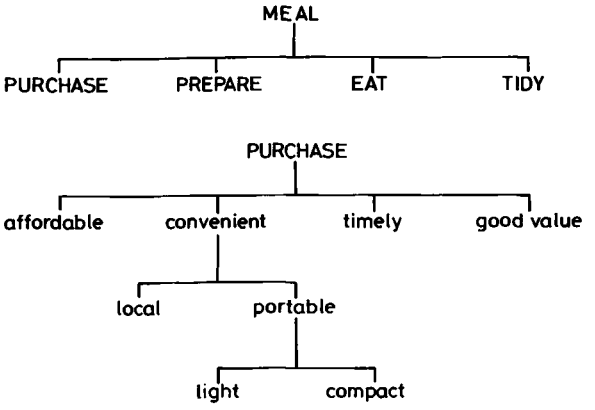
Fig. 2 D2S2 specification of *ICL Technical Journal* structure

specified. To begin with, identify the functions that have to be performed: these are taken to be PURCHASE, PREPARE, EAT and TIDY. Now consider the attributes of each of these functions. For PURCHASE, one might wish that the purchase of food would be within our means, good value etc. Some of these attributes may require subdividing before a sensible definition of them can be given. To quantify an attribute it is necessary to define a measuring concept (e.g. 'time'), a measuring method (e.g. 'use a stopwatch') and the tolerable range of values. The values may be specified as: *worst* (poorest value tolerable), *planned* (expected value) or *best* (optimistic but realistic value). Fig. 3 shows a partial explosion of the specification for having a meal. The specification could be given to another person to fulfil parts of it (not the eating!) in the knowledge that one would get exactly the service required.

4 Semiformal methods

4.1 Programming languages

Programming languages have been used for the specification of algorithms for



portable

definition : the ease with which the foodstuffs can be transported
explosion : see 'light' and 'compact'

light

definition : ease of lifting foodstuffs
measure : weight
method : scales
worst : 10 kg
planned : 0·5 - 1kg
best : 0·25 kg

compact

definition : ease of carrying foodstuffs
measure : longest dimension when packed
method : ruler
worst : 30 cm
planned : 15 - 20 cm
best : 10 cm

Fig. 3 DBO partial specification of a meal at home

some time. There are many examples of ALGOL, and more recently PASCAL[9], used in this style of specification. It has been argued that programming languages are really implementation languages and hence not suitable for abstract specification. The main argument in favour of programming languages in this role is that a high-level language should be machine-independent and so appropriate for specification. Unfortunately, most conventional languages make it impossible to avoid building in some implementation bias (e.g. trading store against processing time). The best approach seems to come from very high level languages such as those employed in *functional programming*[10]. It is possible to give the elements of a programming language a precise mathematical meaning, or *semantics*. A programming language treated in this way can be regarded as formal. However, most examples of specifications using programming languages rely on the informal understanding of the reader.

A typical functional language[10] allows atomic values (e.g. 10 and FRED), lists (e.g. (ALPHA BETA GAMMA)), variables (e.g. x), expressions (e.g. length(side) +5) and function definitions (e.g. square $(y) \equiv y*y$). An empty list is denoted by the atom NIL. Built-in functions include eq (equality), car (to extract the first item of a list), and cdr (to produce a list with its first item removed).

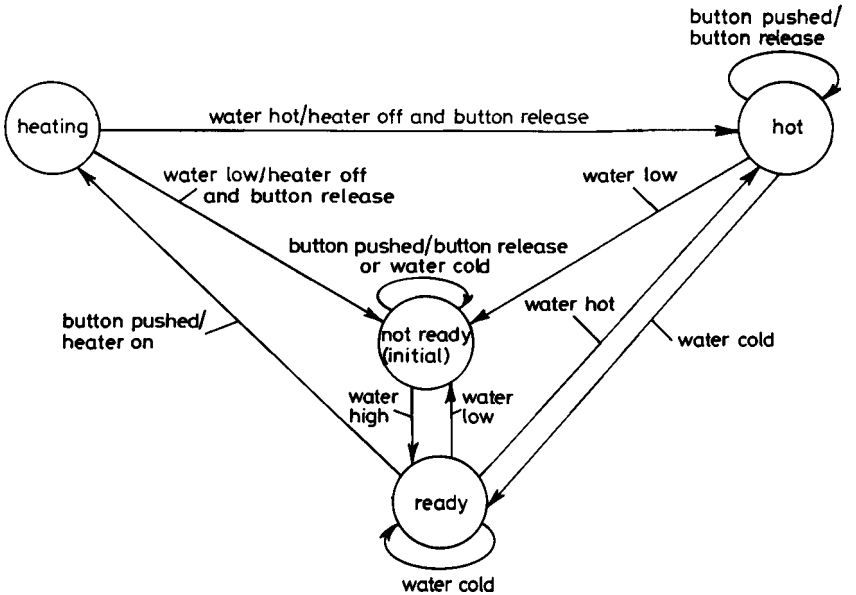
Suppose that a simple calculator to add or multiply a list of numbers is required. The calculator is to be given its input as a list like (SUM 31 3 19) or (PRODUCT 5 2 9) and is expected to output 53 or 90, respectively. The specification of such a calculator might be:

```
calculator(list) ≡ if eq (list,NIL)
                     then NIL
                     else if eq(car(list),SUM)
                            then sum(cdr(list))
                            else if eq(car(list),PRODUCT)
                                   then product (cdr(list))
                                   else list

sum(list)        ≡ if eq(list,NIL)
                     then 0
                     else car(list)+sum(cdr(list))

product(list)    ≡ if eq(list,NIL)
                     then 0
                     else car(list)*product(cdr(list))
```

4.2 State machines

State machine techniques stem from the theory of *finite state automata*[11]. This is closely related to the theory of grammars (Section 5.2) and nets (Section 5.4). However, the widespread use of state machine descriptions by people unfamiliar with the theory shows that the underlying mathematics does not need to be visible. A state machine consists of a number of *states* between which *transitions* are possible. The machine exists in one state at a time, and is triggered to move to a new state by some *event*; this transition to a new state may be accompanied by an *action*. *Extended finite state machines* allow for a number of *state variables* in addition to the basic machine state. The basic machine state describes the gross behaviour of the machine, and the state variables describe the finer details.

Graphical and tabular notations are both used for state machines. The precise notations used vary somewhat; the following description is based on the way in which ICL has specified some communications protocols. In the graphical notation, states are shown as circles and transitions as arcs between them. States are named, and transitions are labelled with their event and action; a null action is omitted. In the tabular notation, states are shown as columns and events as rows in a matrix. Each entry in the table shows the resultant action and the



States / Events	not ready (initial)	ready	heating	hot
button pushed	button release	heater on	X	button release
	-	heating		-
water hot	X	-	heater off and button release	X
		hot	hot	
water cold	-	-	X	-
				ready
water high	-	X	X	X
	ready			
water low	X	-	heater off and button release	-
		not ready	not ready	not ready

Notation

state

event/action

action
new state

- null action or no state change

X impossible

Fig. 4 State machine specification of hot-water urn

resultant state. A dash is used to indicate a null action or no state change. If an event is impossible in a given state, a cross is shown in the entry; this implies that some implementation-dependent error action must be taken.

The example of a state machine given in Fig. 4 is that of a hot-water urn. The urn has a button which is pushed to heat the water; when the water is hot the button is automatically released. The urn has a thermostat and a water-level indicator to prevent damage. To get the feel of the method, try following the paths through the diagram or table. It is instructive to try out some of the more unusual sequences like emptying the urn while it is heating, or filling it with hot water.

5 Formal methods

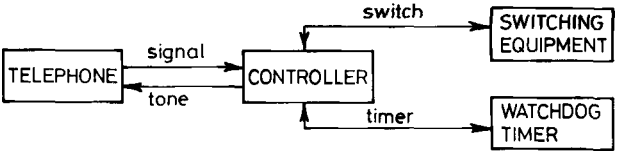
5.1 Algebras

A whole family of algebraic specification methods exists. Note that this is not the kind of algebra which is taught at school, but can be thought of as a generalisation of it. An algebraic specification technique defines a set of symbols, operators for combining symbols, and laws which describe the properties of the operators. Some algebraic techniques are better suited for specifying static characteristics, such as the properties of data structures or sequential systems. Others are better suited for specifying dynamic characteristics, such as the behaviour of concurrent systems (i.e. those with components which operate in parallel). Only the latter category of specification technique will be discussed in what follows.

The common theme in algebraic methods for specifying concurrency is that a system can be thought of as a collection of *processes*. These processes may communicate along *channels*. The processes run independently except when they have to communicate with others. A multilevel description is also possible, with processes at one level being viewed at a lower level as a collection of communicating smaller processes. It is possible to give general algebraic rules about the ways in which processes behave and can be combined.

Two of the algebraic specification languages for concurrency which have been developed in the UK are CSP[12] (Communicating Sequential Processes) and CCS[13] (Calculus of Communicating Systems). Both have been well researched and have been applied to a variety of problems[14,15]. To illustrate the approach a simple CSP example will be given, omitting the more complex features of the language.

A CSP process can be specified precisely by giving its *traces*, i.e. a description of all the sequences of messages which can pass along its channels. The expression c!m means that message m is transmitted on channel c; similarly c?m is used to indicate reception of a message. Each step in the unfolding of process behaviour is separated by the arrow symbol. At some point a choice of behaviour may be




```
RECORDED ANNOUNCEMENT SERVICE
   = TELEPHONE || CONTROLLER || SWITCHING EQUIPMENT || WATCHDOG TIMER

CONTROLLER   =  DISCONNECTED

DISCONNECTED =  signal?off-hook → tone!dialling →
                (   signal?on-hook → DISCONNECTED
                 □  signal?number → switch!number →
                       timer!start → CONNECTING)

CONNECTING   =     signal?on-hook → switch!abandon →
                       timer!stop → DISCONNECTED
                 □  switch?waiting  → timer!stop →
                       tone!ringing  → RINGING
                 □  switch?engaged  → timer!stop →
                       tone!engaged → CONNECTING
                 □  switch?unobtainable  → timer!stop →
                       tone!unobtainable → CONNECTING
                 □  timer? expired → switch!abandon →
                       tone!unobtainable → CONNECTING

RINGING      =     signal?on-hook → switch!disconnect → DISCONNECTED
                 □  switch? connected → CONNECTED

CONNECTED    =     signal?on-hook → switch!disconnect → DISCONNECTED
                 □  switch?voice → tone!voice → CONNECTED
```

Notation

c!m	transmit message m on channel c
c?m	receive message m from channel c
→	followed by
□	alternative operation
‖	parallel operation

Fig. 5 CSP partial specification of recorded announcement service

possible: in this case the alternatives are separated by the box symbol. Finally, parallel running of processes is allowed: the parallel bar symbol is used to join concurrent processes into a single larger process.

Using this compact notation it is possible to describe a wide variety of systems. By way of example, Fig. 5 gives a CSP description of telephoning a recorded announcement service. The complete telephone system (seen from one side) consists of the calling telephone, the controlling unit in the telephone exchange, the telephone switching equipment and a watchdog timer to deal with equipment failure. Only the exchange controller process is described in detail. This starts off in the disconnected state and then unfolds through various sequences

of events. Try following the example: it is not difficult to understand given a knowledge of how telephones usually behave.

5.2 Grammars

The theory of grammars is very closely related to that of algebraic specifications (Section 5.1) and state machines (Section 4.2) but is usually studied as a separate subject. Once again, this is not the schoolroom meaning of grammar but is a general mathematical treatment of the same concepts. Grammars concern themselves with the syntax rules of a stream of symbols, i.e. the rules which govern in what order the symbols can be written.

Many varieties of grammars exist, some quite exotic. Most programmers will be familiar with BNF (Backus Naur Form), originally used to define Algol 60[16] but subsequently employed in many variants to describe other languages. The example below uses some of the features of the British Standard[17] style.

The form of a language can be specified as a set of *syntax rules*: these show how symbols in the language can be expressed as other combinations of symbols. Some symbols are basic to the language and are written in double quotes. Normally, however, symbols are defined in equations of the form 'symbol = . . .' The right hand side of the equation may list alternative definitions, separated by the symbol |. Optional parts of a definition are shown in square brackets [ ], and repeated parts (zero or more repetitions) are shown in curly brackets { }. A series of parts forming one alternative definition is separated by commas.

Using this simple subset of the full notation it is easy to specify the syntax of simple expressions. For example, the definition of floating-point numbers such as 3·14159 or −0·618 might be:

```
number = [sign], digits, [point, digits]
sign   = "+" | "-"
digits = digit, {digit}
digit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
point  = "."
```

5.3 Axiomatic methods

Axiomatic methods specify the behaviour of a system by asserting *axioms* about its initial and final states, i.e. general properties which a system must satisfy. Two of the axiomatic specification techniques used in the UK are the Rigorous Software Development approach[18] and Z[19]. The Z technique is straightforward and has a simple mathematical basis; only some of the basic concepts are described in the following example.

Z is based on the concept of a *schema*, which is shown as a box divided horizontally into two parts. The box is labelled with the schema name in the top left hand corner. The upper part of the schema box defines the inputs and

outputs as a set of variables. Each variable is named and given a *type*, which simply represents the set of permissible values. For example, 'e : EVENS' might declare a variable which was one of the set of even numbers. The lower part of the schema box defines the relationships, or *predicates*, which must hold between the variables. Because operations frequently change the state of a system, it is convenient to refer to the variable values which result from an operation by adding a single quote to their name, e.g. e′. Predicates expressing input/output properties generally state the relationship between the unprimed and primed values. A schema name may be used as an abbreviation for its upper and lower parts, and schema names may also be used with a single quote to denote their results. When a schema name is used inside another schema, it is understood that the two parts of the schema are added to those of the schema in which it is mentioned.

This notation is illustrated by the example of a cashpoint machine. It is first assumed that a current account is represented by total deposits and withdrawals to date and the resulting balance; the obvious relationship between these holds. This is shown in the first schema of Fig. 6. A cashpoint withdrawal is specified by relating the amount required and the state of the account before and after the withdrawal attempt. The second schema of Fig. 6 expresses the fact that the withdrawal is forbidden if it would take the account into the red; otherwise the account is updated by the amount of the withdrawal.

5.4 Net theory

Net theory has close affinities with algebraic methods (Section 5.1) and state machines (Section 4.2), but has a well developed culture of its own. A system is viewed as a set of potential states with possible transitions between these. The actual global state is indicated by placing *tokens*, or markers, in potential states so as to show which conditions hold. A graphical representation is generally used, and the resultant network of states and transitions gives the specification technique its name. Net theory has been applied to many problems and has developed into a number of specialised varieties. It has been used on a number of systems to show that they are *live* (i.e. will not lock up) and *safe* (i.e. will always satisfy certain properties).

Petri Nets[20] are simple forms of state transition networks which have been well studied. Potential states are shown as circles and transitions are shown as boxes. The tokens are shown by placing a dot in a potential state. Arrows point from states to transitions and transitions to states to show the structure of the system.

The example in Fig. 7 is for a simple communications protocol with handshaking on each data transfer. The source process receives input messages and passes them via a line to the sink process for output. The Petri net shown is quite easy to follow using small counters to represent the tokens; the initial position of the tokens is in the *waiting for input* and *waiting for receive* states. Transitions may occur only when all the states leading to a transition hold tokens. When this happens, the tokens in all the input states disappear, and tokens in all the output

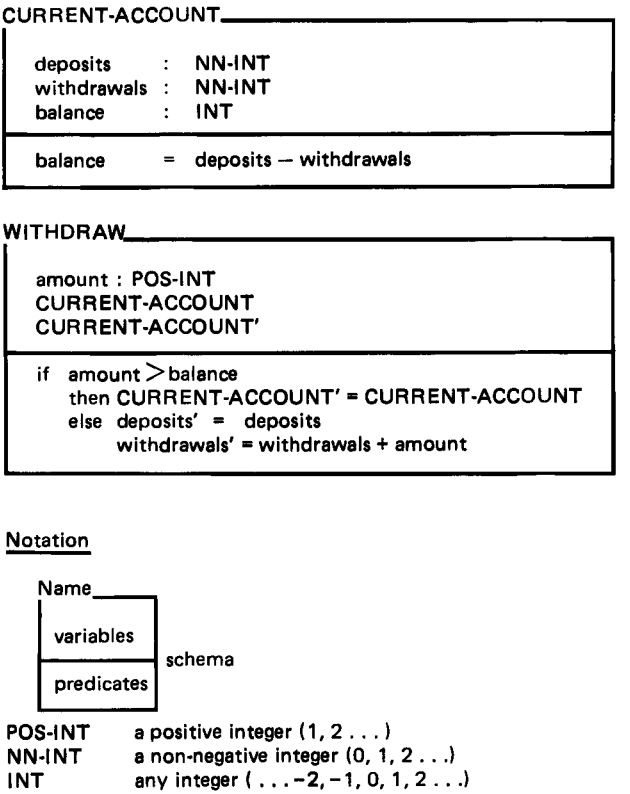


Fig. 6 Z description of cashpoint withdrawal

states appear. Try playing the 'token game' with Fig. 7 by starting with an input message (i.e. letting the *waiting for input* to *waiting to transmit* transition occur).

5.5 *Temporal logic*

Mathematical logic concerns statements of truth or falsity plus rules of deduction. Propositional calculus[21] is the simplest form of logic. It operates with *propositions*, which are assertions that are true or false (e.g. 'the author of this paper is a Martian'), and with *connectives*, which expresses relationships between propositions (e.g. 'it is raining' implies 'the streets are wet'). Predicate calculus[21] extends propositional calculus by allowing *predicates*, which are general assertions about variables (e.g. 'X is red'), and *quantifiers*, which express properties of the whole set of things being considered (e.g. 'there is some Y such that . . .'). Finally, temporal logic[22] adds predicates over time (e.g. 'the light is on *now*') and *temporal operators* (e.g. 'henceforth it is the case that . . .'). Temporal logic can be thought of as a kind of axiomatic method (see Section 5.3) but is usually treated as a separate method.

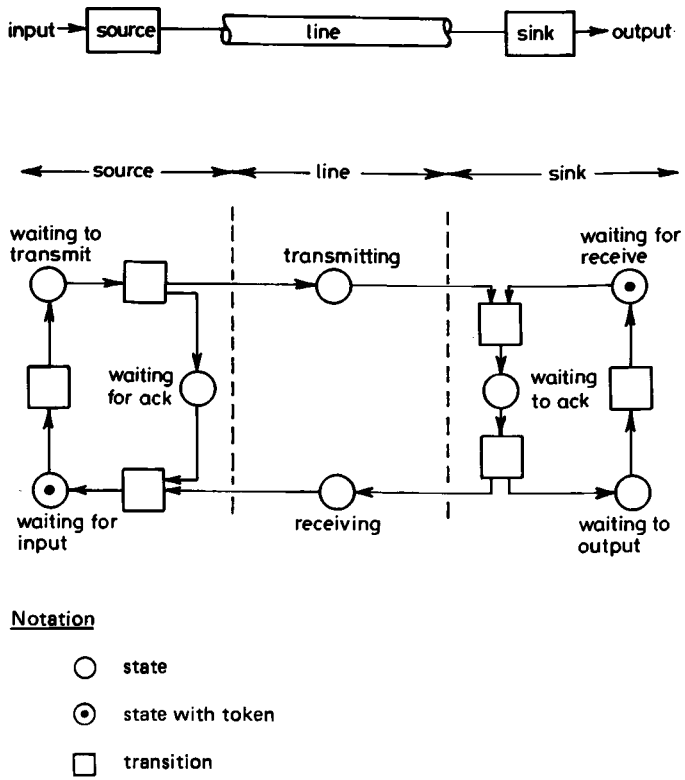


Fig. 7 Net theory description of handshaking protocol

Many flavours of temporal logic are used, but a particularly simple one will be taken as illustrative. This employs two temporal operators: a square □ indicates that the following statement will *henceforth* hold, and a diamond ◇ indicates that the following statement will *eventually* hold. A highly desirable property of restaurants is that once they are open one will ultimately be served. This can be specified as:

$$\text{OPEN} \Rightarrow \Box\,(\text{WAITING} \Rightarrow \Diamond\,\text{SERVED})$$

where the arrow symbol means "implies".

6 ICL involvement

ICL has considerable experience in most of the specification methods described in this paper. Among the informal methods, ICL is actively using Gilb's technique[2] to clarify project and corporate goals. Another major piece of ICL work that deserves mention is CADES[23] (Computer-Aided Design and Evaluation

System). This is a design system which allows, among other things, a structural model of a system to be developed and manipulated in a database. The model reflects the hierarchical breakdown of data and functions within the system. CADES was used to develop ICL's operating system VME (Virtual Machine Environment). Among the semiformal methods, the use of state machines will be mentioned. ICL has used this technique to specify and implement many system components, including some major communications protocols such as X25[24] and ECMA-72[14].

Formal methods offer the best long-term hope for precise and verifiable specifications. ICL, of course, uses grammars in compiler and language design. A variety of system components have been specified using CSP[12], CCS[13] and the Rigorous Software Development method[18]. CSP and Z[19] are currently being used to specify facilities such as communications services and protocols, input/output systems, and database systems. Work on tools to support specification methods has also begun. ICL is collaborating with the leading UK universities on specification techniques.

7 Conclusion

A classification of specification methods has been presented. Examples of methods in each of the informal, semiformal and formal classes have been given. Most of these methods have been used in ICL, some quite extensively. There are two clear future trends in specification methods: the formal ones are becoming more popular, particularly in industry; and a general shaking out and unification of methods is taking place. The future seems to hold two main approaches: transition methods (algebras, grammars, nets, state machines etc.) and assertion methods (axiomatic systems, temporal logic etc.).

Specification techniques are rewarding to study and are passing through a very interesting phase at the moment. It is hoped that the reader has learned that plain English is not the only approach and that there really is a way forward towards better specifications.

Acknowledgments

I am grateful to the colleagues who have supported my work and gently educated me out of my ignorance. Special mention must be made of Graham Pratten who first introduced me to formal methods, and of John Brenner for making possible the QUASARS project, which is studying specification techniques. My thanks also go to Alastair Tocher and Bob Snowdon for commenting helpfully on a draft of this paper.

References

1 COLLINS: 'Concise English dictionary, Collins, London, 1982.
2 GILB, T.: *'Design by objectives'*, North Holland, Amsterdam, (in press).

3 SMITH, W.: 'Software quality assurance', Invited ICL Lecture, London, November 1983.
4 HILL, I.D.: 'Wouldn't it be nice if we could write computer programs in ordinary English', *The Computer Bulletin*, June 1972, 306-312.
5 NAUR, P.: 'Formalisation in program development', *BIT*, 1982, **22**, 437-453.
6 BSI: 'Data processing program and data flow chart symbols, rules and conventions', Standard BS 4058, British Standards Institute, London 1973.
7 MacDONALD, I.G. and PALMER, I.R.: 'System development in a shared data environment', in *'Information systems design methodologies'*, OLLE, T., SOL, H.G., and VERRIJN-STUART, A.A. (Eds.), North Holland, Amsterdam, 1982, 235-283.
8 IBM: 'HIPO – a design aid and documentation technique', Manual GC20-1851, International Business Machines Corp., Research Triangle Park, North Carolina, 1975.
9 ECMA: 'Local area networks – CSMA/CD baseband link layer', Standard ECMA-82, European Computer Manufacturers Association, Geneva, 1982.
10 HENDERSON, P.: *'Functional programming – application and implementation'*, Prentice-Hall Int., Eaglewood Cliffs, New Jersey, 1980.
11 HOPCROFT, J.E. and ULLMAN, J.D.: *'Formal languages and their relation to automata'*, Addison-Wesley, Reading, Massachusetts, 1969.
12 HOARE, C.A.R.: 'Notes on communicating sequential processes', in *Monograph PRG-33*, Programming Research Group, Oxford University, 1983.
13 MILNER, R.: 'A calculus of communicating systems', in *Lecture notes in computer science 92*, Springer-Verlag, Berlin, 1980.
14 MAPSTONE, A.S.: 'Specification in CSP language of the ECMA-72 Class 4 transport protocol', *ICL Tech. J.*, 1983, **3**, (3), 297-312.
15 SHIELDS, M.W., and WRAY, M.J.: 'A CCS specification of the OSI network service', (Internal Report CSR-136-83), Department of Computer Science, Edinburgh University, 1983.
16 NAUR, P. (Ed.): 'Revised report on the algorithmic language ALGOL 60', *Comm. ACM*, 1963, **6**, 1-17.
17 BSI: 'Method of defining syntactic metalanguage', Standard BS 6154, British Standards Institute, London, 1981.
18 JONES, C.B.: *'Software development – a rigorous approach'*, *Prentice-Hall Int.*, Eaglewood Cliffs, New Jersey, 1980.
19 SUFRIN, B.A.: 'Formal system specification – notation and examples', in *'Tools and notations for program construction'*, NEEL, D., (Ed.), Cambridge University Press, 1982.
20 JANTZEN, M. and VALK, R.: 'Formal properties of place/transition nets', in *'Lecture notes in computer science 84'*, Springer-Verlag, Berlin, 1980.
21 LIPSCHUTZ, S.: *'Set theory and related topics'*, McGraw-Hill, New York, 1964.
22 HAILPERN, B.T.: 'Verifying concurrent processes using temporal logic, in *'Lecture notes in computer science 129'*, Springer-Verlag, Berlin, 1982.
23 SNOWDON, R.A.: 'CADES and software system development', in *'Software engineering environments'*, HUNKE, H., (Ed.), North Holland, Amsterdam, 1981, 81-95.
24 TURNER, K.J.: 'Designing for the X.25 telecommunications standard', *ICL Tech. J.*, 1983, **2**, (4), 340-364.

Solution of the global element equations on the ICL DAP

A. McKerrell
Department of Applied Mathematics and Theoretical Physics, University of Liverpool
L.M. Delves
Department of Statistics and Computational Mathematics, University of Liverpool

Abstract

The global-element method is a variant of the finite-element method for the solution of elliptic partial differential equations which uses a few very-high-order elements; because of this, its implementation involves a great deal of regularity and has considerable inherent parallelism. As part of a study of parallel algorithms for PDEs we have implemented the two-dimensional GEM program on the ICL 2980 computer at Queen Mary College and then transferred the solution phase of the program to the DAP which is embedded in that system. We report here on the results obtained.

1 Introduction

There is now considerable experience of the use of the distributed array processor (DAP), on a wide variety of problems; some of this is reported briefly in Reference 1. In particular, it has been evident from the initial design stages that architecturally the DAP is well suited to the solution of partial differential equation problems by finite-difference techniques because the nearest-neighbour connectivity reflects quite well the local approximations made in that method. We report here on work undertaken on an alternative method for solving PDEs, the global-element method[2,3]. This method is a variant of the finite-element method; it differs from the conventional FEM in using a few elements with a high- (and variable) order approximation in each, defined in terms of a Chebyshev polynomial expansion within each element, the polynomial being in a set of suitably mapped co-ordinates. The algorithm which results is interesting from the DAP viewpoint; it contains a high degree of inherent parallelism but without the obvious local connectivity of the DAP processor matrix. It is therefore not obvious how well the DAP can exploit the parallelism, i.e. how well the GEM algorithm will perform on the DAP.

We have underway a research programme designed to answer this and other questions:

- what algorithms best implement the basic GEM method on a parallel processor architecture such as that of the DAP?

- how does the DAP compare with other available machines for this type of problem?
- how well does the global-element method compare with a fixed-order finite-element method or with a nested dissection approach?

In a previous paper[4] preliminary results were discussed for a one-dimensional GEM program. These results were encouraging; and in this paper we report our experience in adapting the full two-dimensional GEM program (GEM2) for the DAP. This program, like any other general PDE solver, is a quite substantial piece of code; the conversion exercise gives some indication of the effort needed to adapt an existing code for the DAP, given that no attempt is made to carry out any major rethinking of the algorithms used. Since it is often said that algorithms *should* be completely rethought for the DAP, the results are also of interest in indicating what performance gains can be made *without* such a rethink; we do not yet know what further gains might be made if we did rethink the algorithms used.

The program GEM2 is described briefly in Reference 5. It provides facilities for specifying an arbitrary region, differential equation and boundary conditions, and for subdividing the region into a number m of elements. It then proceeds in two stages. In the first stage the linear equations describing the approximate solution are constructed in a special condensed form. With m elements and an approximation of order n in each element there are mn^2 equations in mn^2 variables; the corresponding matrix may be partitioned as an $m \times m$ set of blocks, with each block being $n^2 \times n^2$. The blocks are either full or empty; diagonal blocks are full, whereas off-diagonal blocks are empty unless they correspond to two elements which have a side in common. Off-diagonal blocks also have a relatively simple structure (they are of low block rank), which implies that they can be produced relatively cheaply using fast Fourier transform techniques and that the information they contain can be stored in coded form in order (n^2) rather than order (n^4) locations[3]. FFT techniques are used also to compute the essential information required for the assembly of the diagonal blocks in order $(n^2 \log(n))$ operations, although filling the n^4 entries requires order n^4 arithmetic operations.

The result of the matrix-assembly phase is a block sparse matrix in partly coded form. There is considerable structure in the matrix, and it is possible to develop rapidly convergent iterative methods for the solution of the linear equations defined by the matrix and right hand side, which for large m,n have an operations count of order mn^4.[4,6] Unpublished work suggests that even lower operation counts may be achievable. However, the current production version of GEM2 does not make use of these techniques, but incorporates a disc-based direct solver with an operations count of order (mn^6). For large n, and especially for problems with relatively large m, the solution phase of the program both dominates the time taken and limits the problem size which may be tackled.

The performance of GEM2 on a variety of problem types has already been reported[7,8,9]. The results suggest that the techniques used do indeed provide a

Table 1 Overall times for matrix setup on several systems

m	System	n=4	n=6	n=8	n=10	n=12	n=14	n=16
1	ICL1906S	2·4	3·2	11	15	23		
1	CDC7600	1·7	6·1	231	270			
1	ICL2980	2·0	2·8	7·4	11	17	27	61
6	ICL1906S	15	20	73	103			
6	CDC7600	29	55					
6	ICL2980	11	16	50	74	109	166	406

m is the number of elements; n the order of approximation in each element. Times are in s.

Table 2 Overall times for matrix solution on several systems

m	System	n=4	n=6	n=8	n=10	n=12	n=14	n=16
1	ICL1906S	0·2	0·7	2·4	6·5	15		
1	CDC7600	0·4	2·1	7·5	20			
1	ICL2980	0·6	1·1	3·0	7·7	18	38	76
1	DAP, R*8	0·8	1·2	1·8	3·8	7·8	14·3	16·4
1	DAP, R*4	0·6	0·7	1·0	1·9	3·6	6·2	7·8
6	ICL1906S	4·2	23	94	309			
6	CDC7600	30	164					
6	ICL2980	4·9	25	120	421	1278		
6	DAP, R*8	7·6	18	42	130	308	574	1095
6	DAP, R*4	5·8	14	34	90	199	386	819

Notation as in Table 1. Times for the DAP include both 2980 (host) and DAP mill times

method which is rapidly convergent as the order of approximation n is increased, and that GEM2 performs well on a fairly wide class of problems. We are interested here, however, not in convergence rates but in speed: how fast does the program run, in a serial environment and on the DAP? GEM2 is written in Algol 68; Tables 1 and 2 give an indication of its performance on several conventional serial machines (ICL 1906S; CDC 7600; ICL 2980) for which Algol 68 compilers are available. Results are given for two model problems using one and six elements to describe the region and for various values of n. The upper limit for n indicates the maximum which could conveniently (i.e. without a great deal of extra work) be run on the relevant machine. This limit is set either by the amount of main store available or (more usually) by the run times needed for larger n. Both model problems require the solution of Laplace's equation over a square, with smooth (m=1) or discontinuous (m=6) boundary conditions; they are described in Reference 8 (examples 1 and 9).

The relative performances of the machines are discussed later (see Section 3). The results show clearly the main characteristics of GEM2 on conventional serial machines:

- The matrix setup time is roughly proportional to the number of elements m. For small m solution times increase somewhat faster than this but are expected to be proportional to m for large m.
- For large n the setup times increase[3] roughly as $n^2\log(n)$; the solution

times increase as n^6. For m=6 and n>=12, the setup times are less than 10% of the total time taken. In converting GEM2 for the DAP we have therefore looked at the solution phase first.

2 Conversion for the DAP

The solution phase as programmed in the current production version of GEM2 treats the $mn^2 x mn^2$ matrix involved as an mxm matrix of $n^2 x n^2$ blocks, most of which (for large m) are empty. The setup phase is the part of the program which generates these blocks; however, in GEM2, the blocks are not produced directly. Rather, for reasons of efficiency in both time and space, the setup phase of GEM2 produces a coded and compressed form of the blocks; specifically, it produces a set of n vectors u_k, v_k, w_k, y_k, and of $n \times n$ matrices A_k, B_k. The vectors depend on the form of the expansion set used to define the approximate solution; the matrices, on the coefficients in the differential equation[3]. The suffix k labels the vectors and matrices; k runs from 1 to an upper limit which is only slightly problem dependent and which is in any event independent of the parameters m, n. The relationship between these vectors and matrices and the $n^2 \times n^2$ blocks is given by eqn. 1 below.

Using this compression technique, the storage needed by the setup phase is only of order mn^2, rather than the mn^4 which would be needed to hold the fully assembled blocks. As noted in the previous section, the (serial) operations count for the setup phase is only of order $n^2 \log(n)$ using the FFT techniques described in Reference 3.

The solution code in GEM2 therefore has two major components, which we consider in turn.

2.1 Matrix expansion

The compressed form of the matrix produced in the setup phase is first expanded to normal matrix form. This expansion first represents the differential operator involved and then adds the boundary and interelement interface conditions, which form a contribution to the diagonal blocks and represent the whole of the nonzero off-diagonal blocks. For a given block, the requirement is to form an $n^2 \times n^2$ matrix L with elements of the form[3]

$$L_{pq,\,rs} = \sum_{k=1}^{x_{max}} u_{kq} * v_{ks} * A_{kpr} + \sum_{k=1}^{y_{max}} w_{kp} * y_{kr} * B_{kqs} \text{ (usually)} \quad (1)$$

$$= 0 \text{(sometimes, depending on the values of the pairs } (p,q) \text{ and } (r,s))$$

Table 3 Times, in s, for suboperations in the matrix solution code

n	Operation: Time on	matrix expansion	$A:=A-B*C$		solve block	complete solution
4	2980	1·1	0·9		0·6	4·6
	OCP	0·6	0·3		0·23	3·1
	DAP	1·4		3·2		4·6
6	2980	4·7	9·4		4·9	24·4
	OCP	1·4	1·0		0·7	8·6
	DAP	2·1		7·2		9·3
8	2980	18·6	61		28	187
	OCP	3·4	3·7		2·3	27
	DAP	2·8		12·9		15·8
10	2980	38	242		111	434
	OCP	8·1	9·0		5·5	65·8
	DAP	3·5		60·8		64·3
12	2980	78·8	785		357	1316
	OCP	15·0	18·9		15·4	134
	DAP	4·2		170		174
14	2980	-	-		-	-
	OCP	27·9	35·6		27·0	269
	DAP	4·9		300		305
16	2980	-	-		-	-
	OCP	64·7	61·6		47·2	588
	DAP	5·5		501		507

Problem: Motz problem (see Reference 7) with $m=6$
2980: times for operations running on the 2980 in Algol 68
DAP: mill time on the DAP (DAP Fortran) in REAL*8
OCP: residual 2980 times, after conversion to the DAP

In eqn. 1 u_{kq} refers to the qth element of the vector u_k, and similarly for v_{ks}, w_{kf} and y_{kr}. A_{kpr} and B_{kqs} are the pr and qs elements of the matrices A_k and B_k; and the suffices p,q,r and s run from 1 to n.

The expansion in eqn. 1 takes a significant proportion of the total time (see Table 3). The serial implementation involved a straightforward coding of eqn. 1 in which each sum of the form of eqn. 1 was performed in turn. Considering for simplicity only the first of the two summations, the code had the structure shown in Fig. 1*a*. A possible first suggestion is to transfer the k loop to the DAP, as indicated in Fig. 1*a*. This is very straightforward, but fails to utilise the processors very effectively because this loop is very short, with a length (x_{max}: see eqn. 1) which is independent of m and n. However, reordering the loops as in Fig. 1*b* produced two advantages:

- the serial code with reordered loops ran almost twice as fast as before (which made us feel a little foolish, if gratified)

– we could now transfer the r,s loops directly to the DAP as indicated by the DAP1 markers in Fig. 1*b*, giving a potential parallelism of order n^2.

Fig. 1 Structure of the code for eqn. 1

a

```
Initial structure

For p,q
Check (p,q); skip if necessary

Check (r,s); skip if necessary

L_pq,rs := 0.0

********DAP START?
For k
L_pq,rs := L_pq,rs + u_kq*v_ks*A_kpr

End k-loop

********DAP END?

End pqrs loops
```

b

```
Recoded form

L := 0.0
For k
********DAP2 START
For p,q
Check (p,q); skip if necessary

***************DAP1 START

For r
temp:=u_kq*A_kpr
For s

Check (r,s); skip if necessary

L_pq,rs := L_pq,rs

+temp*v_ks
End r,s loops

***************DAP1 END

End p,q loops

********DAP2 END
End k loop
```

Since GEM2 currently runs with n in the range 8-16, we expected a reasonably effective DAP code to result from this strategy. In the event, we found that the overheads required in repeatedly calling the DAP (and converting the storage mode of the data from host to DAP mode) inside the p,q loop, were quite high. The final version of the code included the p,q loops in the DAP code as indicated by the DAP2 markers in Fig. 1*b*. This yields a parallelism of order n^4, provided we can utilise it. The serial implementation of GEM has so far used values of n up to 12. For n up to 16, the $n^2 \times n^2$ matrix involved for each element can be fitted into a DAP supermatrix of the form $M(,,16)$. For fixed k, and ignoring the conditional tests implied in eqn. 1, we are required by eqn. 1 to produce, for the first sum, contributions of the form

$$L_{k,pq,rs} = u_{kq} v_{ks} A_{kpr};\ p,q,r,s = 1, 2 \ldots . n$$

The code written produced the contributions for given p in $M(,,p)$. The columns A_{kp} and vectors u_k and v_k were suitably broadcast into DAP matrices by rows and columns, and then $M(,,p)$ could be generated with two pointwise matrix multiplies. The second sum in eqn. 1 was treated similarly; and the conditionals,

which involve skipping entries with certain values of p,q,r,s, were treated after the sums over k were complete during the necessary copying of L from Fortran Common to the array to be passed back to Algol 68.

2.2 *Solution of the block equations*

The other main component of the solution phase is the solution of the $mn^2 \times mn^2$ set of equations. These are stored as a set of $m \times m$ blocks of size $n^2 \times n^2$; these blocks are stored on disc and buffered in as required by a block Gauss elimination routine.

During elimination two main operations dominate: the reduction of individual blocks to traingular form and the basic operation of a standard row elimination algorithm:

$$A := A - B * C \tag{2}$$

where A, B and C are $n^2 \times n^2$ matrices. It is straightforward to transfer these two operations to the DAP; the Gauss elimination algorithm, used in the serial version to reduce the diagonal blocks, was replaced by a call to a DAP library Gauss-Jordan routine, while the routine used to carry out the operation in eqn. 2 was replaced by a corresponding DAP routine.

Table 2 gives timings for the resulting DAP code, and Table 3 gives results for the individual code sections. All calculations have been carried out in both REAL*8 and REAL*4 arithmetic; the former corresponds to the standard REAL variables in the 2980 Algol 68 compiler. We comment later on the choice of DAP working precision. Several aspects of these results are of interest.

- On any host-attached processor system, the speed improvement is limited by the proportion of code which remains host-resident; if $h\%$ of the host code remains after conversion the speedup factor cannot exceed $100/h$ no matter how fast the attached processor may be. In Table 3 the 2980 entries refer to the mill times taken on the 2980 before conversion to the DAP; those labelled OCP refer to the remnant 2980 mill time after conversion. The results show that, for the sections relating to the operation in eqn. 2 above and solution of individual blocks, the residual host code takes only about 6% of the original times even for n=8, and the proportion reduces rapidly as n increases; potentially very large speedups are possible. The actual speedup ratios (compared with the host 2980) also increase with n.
- Speedup factors for the complete solution phase depend upon m and n as well as on the arithmetic precision used on the DAP. The observed ratios increase with n, as expected; for m=1 and n=16 the ratio obtained reaches 4·6:1 for REAL*8 and 9·7:1 for REAL*4 arithmetic. For m>1, the factors might be expected to be lower than this, since the disc buffering times are significant and cannot be avoided with the current solution algorithm. For m=6 and n=12, the largest value of n for which a direct comparison could be made, the overall factor reaches 4·1:1 in REAL*8 and 6·4:1 in REAL*4 arith-

metic. We were unable to obtain direct measurements for larger n and this number of elements because of the time taken by the serial version; however, the ratio of 6·4:1 compares with 5:1 for n=12 and m=1, so that it would seem that the disc buffering times are outweighed by the extra arithmetic carried out for m=6. This extra arithmetic stems mainly from operations of the form of eqn. 2 above, and are treated very efficiently on the DAP. The speedup for m=6 and n=16 is estimated to be between 9:1 and 13:1, this range depending on how the extrapolation is made.

- Table 2 relates the overall times to those measured on other systems. This table shows that:
  - For the largest value of n for which direct comparisons could be made (n=10 with one element) the DAP REAL*8 version of GEM2 runs approximately 5·3 times as fast in the solution phase as the CDC7600 version; the REAL*4 version about 10·5 times as fast. These ratios increase with m and n. The m=6 runs on the 7600 were abandoned for $n>6$ because of the time taken.
  - The large homogeneous store on the DAP and its higher speed makes it possible to use significantly higher order elements. This is particularly important with the GEM algorithm, since[9] accuracy improves on a 'typical' problem by a factor of ten as n increases by two.

3 Conclusions

The interest in these results lies both in the speed of the converted GEM program and in the ease of the conversion process.

3.1 Speed

The speed of the DAP version depends markedly on the precision of the arithmetic used. We view this here from the engineer's viewpoint. Most PDE applications do not require very high-accuracy solutions, and it seems likely to us that a 'production' version of a PDE program on the DAP would best be set for normal use at REAL*5 (40-bit reals) as giving adequate accuracy at minimum cost. For such arithmetic our results indicate a speedup ratio of about 8·5:1 (m=1) or 7·10:1 (m=6) compared with the ICL2980. These are more or less in line with those reported for other floating-point-dominated applications[1].

Our high ratios relative to the CDC7600 show this machine in an unexpectedly poor light; one would generally rate the 7600 as being significantly more powerful than either the 1906S or the 2980. However, both these machines have very efficient Algol 68 compiler systems; and our figures reflect the relative efficiencies of the Algol 68 compilers on the machines as well as their relative raw powers. That on the 7600 shows up worse in the setup phase, where the code of GEM2 makes heavy use of the advanced structuring facilities of the language, than in the solution phase, where the code is much more straightforward. Unexpected or not, the efficiency of compilation and the overall measured ratios are certainly relevant for the GEM2 package.

3.2 *Conversion costs*

Equally interesting is the scale of the effort needed to achieve these speedup ratios. We have *not* attempted to redesign the algorithms used, nor to carry out any large-scale restructuring of the code. One routine was recoded before conversion; two more replaced by equivalent DAP-Fortran routines (the larger of these was taken from the DAP subroutine library); and, otherwise, routine effort was put into placing data into Common blocks and calling the DAP via the mixed-language facilities on the 2980. This experience is in apparent contradiction to the common belief that it is necessary to rethink algorithms from scratch before the DAP can be utilised effectively. There *is* reason for such a statement; but it applies particularly to the basic operations needed. We have simply made use of a few of the many basic operations for which the rethink has been carried out already, and routines provided in the DAP library. The resulting conversion was in our opinion very cost-effective.

Acknowledgments

The research reported here was funded in part by the US Army European Research Office via Grant No DAJA45-82-C-0010; and by the UK SERC via Grant No GR/A48990 to the University of Liverpool.

References

1 HOWLETT, J., PARKINSON, D. and SYLWESTROWICZ, J.: 'DAP in action', *ICL Tech. J.*, 1983, **3**, (3), 330-344.

2 DELVES, L.M. and HALL, C.A.: 'An implicit matching principle for global element calculations', *JIMA*, 1979, **23**, 223-234.

3 DELVES, L.M. and PHILLIPS, C.: A fast implementation of the global element method', *JIMA*, 1980, **25**, 177-197.

4 HENDRY, J.A. and DELVES, L.M.: 'GEM calculations on the ICL DAP', Proceedings of a Workshop on Vector and Array Processors, Bristol, 1982, (in press).

5 PHILLIPS, C., DELVES, L.M. and O'NEILL, T.: 'GEM2: a program for the solution of elliptic PDEs', Proceedings of a Conference on Tools, Methods and Languages for Scientific and Engineering Computation, Paris, May 1983, North-Holland, (in press).

6 HENDRY, J.A., DELVES, L.M. and MOHAMED, J.: 'Iterative solution of the global element equations', *Comp. Meth. Appl. Mech. Eng.*, 1982, **35**, 271-283.

7 HENDRY, J.A. and DELVES, L.M.: 'The global element method applied to a singular harmonic boundary value problem', *J. Comp. Physics*, 1979, **33**, 33-44.

8 DELVES, L.M., PHILLIPS, C. and O'NEILL, T.: 'A manual of GEM2 examples', Research Report SCM/83/1/1, University of Liverpool, 1983.

9 DELVES, L.M., McKERRELL, A., PETERS, S.A. and PHILLIPS, C.: 'Performance of GEM2 on the ELLPACK problem family', (in preparation).

Quality model of system design and integration

T.L. Faulkner* and M. Small
ICL Mainframe Systems Development Division, West Gorton, Manchester
*now at the ICL Atlas Division, West Gorton

Abstract

In this paper a model is presented relating the quality of a design at each stage of the design and integration processes to the initial parameters of the design and the methods of test used. This model is used to demonstrate the improvements in final design quality which can be expected if integration testing occurs at each level of design decomposition.

1 Introduction

This paper arises from the need to develop computer hardware to a desired level of design quality within an acceptable timescale and cost. This leads to a need to understand the relationships between the various development activities and design quality. The problem of design quality and development cost is becoming more important with the increasing application of VLSI, which permits much greater design complexity for a given product value, but which is costly to develop by cut and try.

The literature contains various models, which have been targetted on software, relating bugs expected to various parameters of the design[1, 2, 3]. Data concerning the observed rates for different kinds of faults found during different stages of development is also available[5, 6].

This paper attempts to produce a structure for understanding this latter information, so as to consider how it may be applied to changing the development methodologies to yield fewer faults. To this end the design and development is assumed to use a top-down structured approach. This can be regarded as a two-stage activity: system design and system integration.
System design is basically a divergent top-down process whereby the system is structured into a hierarchy of partitions with the specified interfaces between

This paper was first presented at the IEEE 13th Annual Symposium on Fault-tolerant Computing, 1983. It is reproduced here with acknowledgment to the IEEE.

partitions. Partitions at the lowest level are then designed independently to meet the requirements of these interfaces. System integration is the convergent bottom-up process of testing a simulated or a real implementation of each partition both independently and then as assemblies of partitions, until the complete system is integrated and tested in the way it will be required to work by the user.

A mathematical model is developed for the process of system integration. This is a Bayesian model which relates the quality of the design at each stage (in terms of the confidence in the absence of design faults or the remaining number of undetected design faults) to the parameters of the initial design and the methods of test. The model is then extended to the design process and used to demonstrate that dramatic improvements in final quality can be expected if system integration testing occurs at each level of design decomposition.

2 Design and development processes

A system design may be regarded as a hierarchy represented by the structure of system specifications illustrated in Fig. 1.

The process of system design is considered to be the partitioning of the design at any level into specifications of partitions and interfaces between partitions. At the lowest level the partitions are the basic building blocks of the design

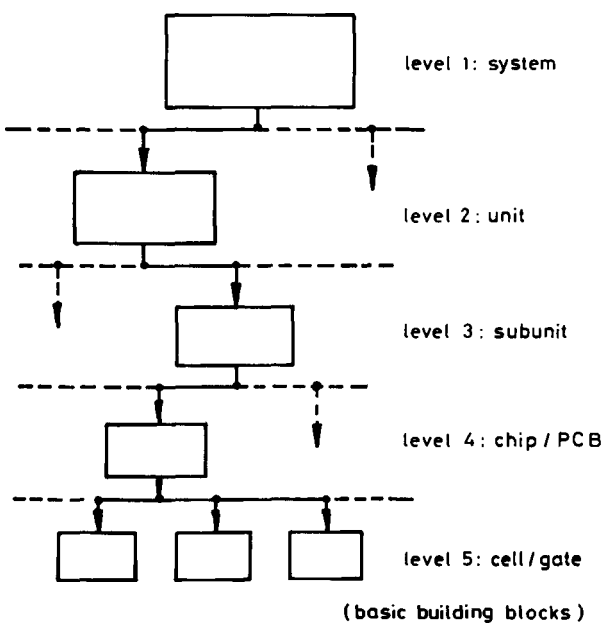


Fig. 1 Hierarchical structure of system design

(which may be gates, cells, MSI ICs, order code instructions etc.) which are assumed to be perfectly specified and without internal design faults. At each level in the system design process design faults in the specifications of partitions and interfaces may occur. At the lowest level only design faults in the building block interconnections are recognised. To correct these faults interconnections and building blocks may be changed or added to, but the set of available building blocks remains unchanged.

The process of system integration is considered to be the progressive assembly of partitions and interfaces, starting from the lowest level until the full system is assembled. This process is regarded as analogous to that of production assembly. During production assembly it is normally accepted that components and assemblies may exhibit mechanical faults which, if left uncorrected, would increase system unreliability. These are monitored and controlled by processes of test and repair at various stages during the production assembly process.

Similar processes are employed during the integration of the design of a new system. Test and design modification processes are introduced to discover and correct faults in the specifications of the design at any level. These design faults are removed by correction of the specification at the highest level at which the fault applies and modification of the lower level designs to agree to this new issue of the higher level specifications. The test is repeated with the changed design and if passed successfully, the integration process is continued. This is illustrated in Fig. 2.

Tests may be organised to validate each stage of the integration process, or may be used in series to give repeated tests at each stage.

The design integration process need not start with the realisation of the final implementation: integration using simulations and hardware or software models

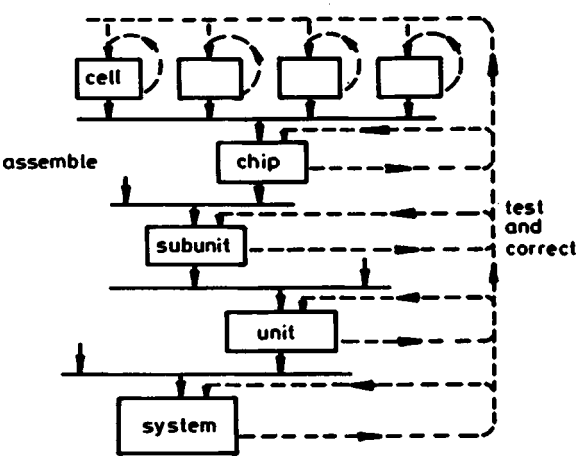


Fig. 2 System integration

of the design or part of it, at any level, may be used with appropriate tests to identify design faults before the final product is built.

3 Concept of quality

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

Faults may be grouped into virtually any classification which is considered to be useful, the only important rules are that each fault class must be essentially independent of all other classes and the list of classes should be complete. This latter requirement is far more difficult to achieve and, in practice, any statement of quality can only be made with respect to some defined list of fault classes. An example of a suitable classification for hardware design faults is given in Reference 5. This data also indicates the various levels of faults which can be expected. To achieve high quality, either the basic processes must be changed to reduce the fault rates or tests must be applied to detect and remove the faults introduced.

4 Representation of quality

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

For fault class j let q_j be the probability of freedom from faults of this class prior to the process or test, and q'_j the probability of freedom from faults of this class among items which have undergone the process or passed the test.

5 Quality and numbers of faults

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

Quality and fault rates are related by a fault distribution. This gives the probabilities that various numbers of faults will occur per class or per product and includes zero faults which is, of course, the ideal quality. The fault distribution may be a set of observed values or be represented by a mathematical model. Two

such models are the single-fault model, in which it is assumed that no more than one fault may occur, and the Poisson model, in which it is assumed that there are an infinitely large number of possible faults, each of which is equiprobable.

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

$$f_j = \ln (1/q_j) \tag{1}$$

where f_j is the average number of faults of class j.

Neither of the above models is completely satisfactory, since faults are not usually equiprobable and, although there may be more than one fault, there is a finite limit to the number of faults. However, considering faults to be Poisson distributed within each fault class gives an acceptable compromise and this is what will be assumed.

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

$$F = \sum_{j=1}^{n} f_j \tag{2}$$

and the probability of no faults:

$$Q = \prod_{j=1}^{n} q_j \tag{3}$$

The above functions, relating quality and number of faults, illustrate how quite small numbers of design faults imply very low design quality (as defined above).

6 Testing and its effect on quality

The quality of items passing a test can be expected to be greater than the quality of those submitted for test providing that the test detects the relevant kinds of faults and is nondestructive. The actual quality of items passing a test depends, for each fault class, upon the quality of items input and the test detection rate for faults of that kind. There are various models which have been described[7, 8] relating post-test quality to pretest quality and test detection. The difference between these models concerns the assumption made regarding the distribution of numbers of faults. Both of the models cited above assume that the test detection rate is the same for all faults and that it is only necessary to detect one of many faults to identify the item as bad. This leads to an increase in the apparent effectiveness of the test when input quality is low and the probability of more than one fault is high.

Assuming faults are Poisson distributed, the quality post test of a given fault class can be derived from Reference 8 to be:

$$\ln q'_j = (1-d_j)\ln q_j \qquad (4)$$

where d_j is the detection rate for fault class j. Also the number of faults post test f'_j is given by

$$f'_j = (1-d_j)f_j$$

7 Sequence of tests

Subjecting some part of a design to a series of tests can improve quality more than any of the tests taken individually can, providing the various tests are independent. Whether or not tests are independent can be difficult to judge; however, the criterion of independence is very simple:

A test may be considered to be independent of another if that test detects the same proportion of faults which are undetected by the other test, as it would detect in the total population of faults. If tests are not independent, then the overall detection of a sequence is, in the worst case, no greater than the highest individual test detection in the sequence.

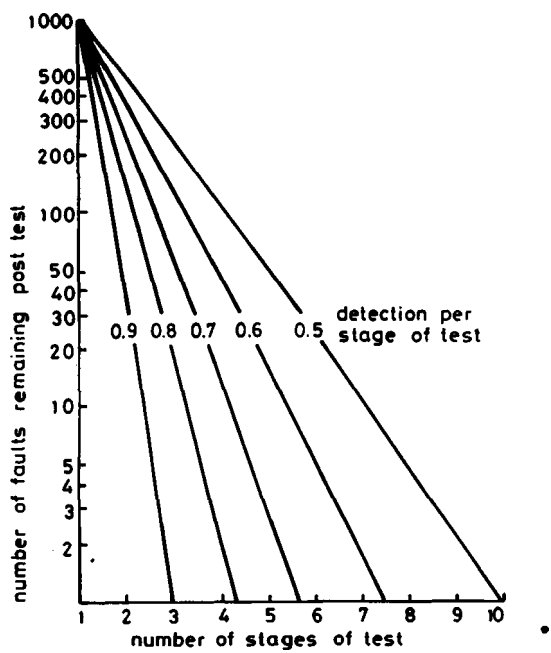


Fig. 3 Sequences of test

Providing tests are independent, the detection of a sequence may be modelled as follows.

Let the detection of the kth test in the sequence be d_{jk}. If there are N_t tests the number of faults post sequence is:

$$f'_j = f_j \prod_{k=1}^{N_t} (1-d_{jk}) \tag{5}$$

i.e. the effective total detection offered by the sequence is:

$$D_j = 1 - \prod_{k=1}^{N_t} (1-d_{jk}) \tag{6}$$

If each test in the sequence has the same detection characteristic (unlikely), the required value for this to obtain a given quality improvement can be seen to be:

$$d_j = 1 - (f'_j/f_j)^{1/N_t} \tag{7}$$

The way in which quality is improved by a sequence of tests, assuming the detection per stage of test is a constant, is illustrated in Fig. 3.

8 Design decomposition

Considering now the design phase, the quality of the overall system expressed in terms of the quality of the specifications at the level of partitioning reached can be seen to decrease if the design is taken to be an assembly of the lower level specifications. The qualities reached at the lowest level of design dominate overall quality at the start of the system integration phase.

What is not clear is the quality at each stage as the decomposition process proceeds. Each of these quality values will depend on the skill of staff in interpreting, expanding and contributing to specification and requirement statements defined at a higher level. Each of these specifications will state the lower level partitions and the functions required from these partitions, and will also need to specify the constraints which implementing the lower levels has imposed on the design.

A *simplifying assumption* may be made that the quality value of the lowest level of design remains approximately constant during the system design process. This shows how the system quality deteriorates during the design decomposition. It is, however, an optimistic assumption, as it implies that the number of faults is a linear function of size. This is to some extent supported by measurements on software[11] provided certain limits are accepted (e.g. small modules).

9 Effects of decomposition on quality

Starting from the lowest level of decomposition, each higher partition is formed by collecting together lower level partitions. If it is assumed that this process detects no faults and introduces none which did not already exist in the lower partitions, the effect of this on quality can be modelled at each level as follows.

First, where several partitions have the same fault class, the quality of the assembly in respect of that class is the product of the item qualities. Secondly, the list of fault classes which must be considered for the assembly must be arranged to include all those applying to the individual items.

Let q'_j be the quality of an assembly of n items in respect of fault class j after assembly but before considering other effects. Let q_{ij} be the quality of the ith item possessing that fault class, then:

$$q'_j = \prod_{i=1}^{n} q_{ij} \text{ and } f'_j = \sum_{i=1}^{n} f_{ij} \tag{8}$$

Since the total decomposition of a system may be large, this can result in a considerable compounding effect on the total number of faults. If there are r levels of breakdown and the expansion at the kth level is n_k the total expansion N is given by:

$$N = \prod_{k=1}^{r} n_k \tag{9}$$

It may be noted that, for a constant expansion per level of n, $N = n^r$, and if the basic elements have equal q_j and f_j the overall qualities Q_j and F_j are:

$$Q_j = q_j^N \text{ and } F_j = Nf_j \tag{10}$$

The way in which overall design quality deteriorates (numbers of faults increase) as the system design is expanded is illustrated in Fig. 4. Here it is assumed that the expansion ratio at each level is a constant.

Consider for example, a system with 100 integrated circuits per PCB, 12 PCBs per subunit, three subunits per unit and three units per system. If it is expected that the original logic design will have one fault per ten integrated circuits the total decomposition quantity will be $3 \times 3 \times 12 \times 100 = 10\,800$. Hence the total number of faults which may be expected is $0{\cdot}1 \times 10800 = 1080$, corresponding to an overall quality of $10^{-4\,3\,5}$.

10 System integration

A common approach to system integration is to assemble, test and correct at

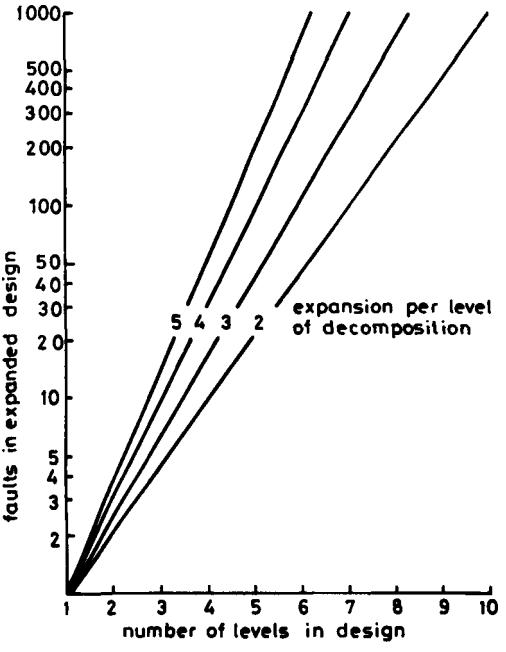


Fig. 4 Design decomposition

each level of recomposition of the design. This modifies the model of collection described above, improving the quality obtained due to the detection of faults at each stage. Assuming independence of tests, the model of collection may be combined with the model of test sequences to give the following model of the process.

At each stage in integration the quality for each fault class is given by:

$$Q'_j = Q_j^{(1-d_{jk})} \quad \text{where} \quad Q_j = \prod_{i=1}^{N} q_{ij} \tag{11}$$

$$F'_j = F_j\,(1-d_{jk}) \quad \text{where} \quad F_j = \sum_{i=1}^{N} f_{ij} \tag{12}$$

If there are $r + 1$ levels in the design, implying r independent test stages in system integration, the total detection of D_j of the integration sequence is:

$$D_j = 1 - \prod_{k=1}^{r} (1 - d_{jk}) \tag{13}$$

Hence the final quality after integration is:

$$Q'_j = Q_j^{(1-D_j)} \text{ and } F'_j = F_j\,(1-D_j) \tag{14}$$

If it is assumed that the quality of each element of the final level of decomposition of the design (q_j, f_j), the expansion ratio (n) at each level and the detection (d_j) at each stage are constant, the final number of faults is given by

$$F'_j = n^r f_j (1-d_j)^r$$

or
$$\ln (F'_j) = r \{ \ln (n) + \ln (1-d_j) \} + \ln (f_j) \tag{15}$$

If the final number of faults F'_j is required to be the same as that in the first level of design (f_j) then:

$$\ln (n) = - \ln (1-d_j)$$

or
$$d_j = (n-1)/n \tag{16}$$

Fig. 5 illustrates the average detection per test stage required with this system quality equal to that of the first level of design. Assuming the expansion ratio per level is constant (n) and the detection required per stage is constant (d), combining Figs. 3 and 4 shows that d is a simple function of n.

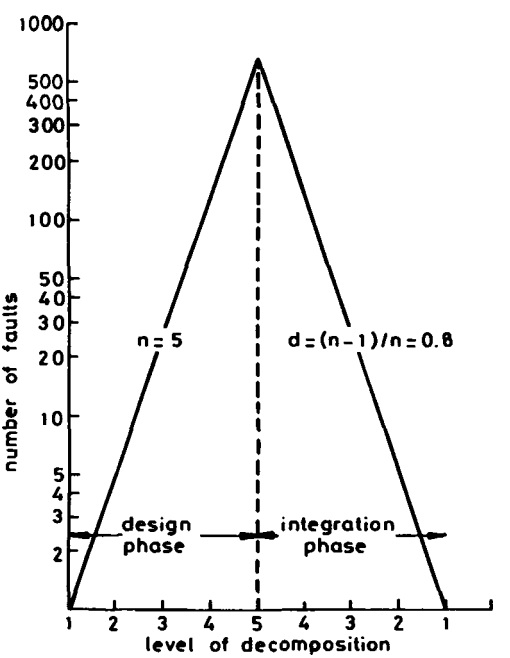


Fig. 5 System integration only after total decomposition

Consider the previous example of a system with a 100 integrated circuits, 12 PCBs per subunit, three subunits per unit and three units per system. If the required design quality is 0·9 the required overall fault detection and detection per stage required may be calculated.

required final fault rate

$$F'_j = \ln (1/0{\cdot}9) = 0{\cdot}105$$

given fault rate $f_j = 0{\cdot}1$ (17)

total expansion $N = 10\,800$ and faults $= 0{\cdot}1 \times 10\,800 = 1080$

Hence the total detection required of the integration sequence

$$D_j = 1 - (0{\cdot}105/1080) = 0{\cdot}9999 \tag{18}$$

and the detection per stage of integration (assuming equal detection per stage)

$$d_j = 1 - \left(\frac{0{\cdot}105}{1080}\right)^{1/4} = 0{\cdot}9 \tag{19}$$

11 System integration at each level of decomposition

If design decomposition proceeds without testing, then all the faults introduced by the process remain to be detected at the recomposition or system integration stage. This is clearly expensive and inefficient, and so various techniques have been developed, which include modelling, simulation and inspection, to allow the design to be checked at each stage of its decomposition. This can be expected to result in an increase in the final design quality, assuming the detection of the extra tests is similar to the detection obtained from the others and that all the tests can be considered independent.

In the extreme case, assume that at each level of decomposition of the design a full system integration is performed (i.e. each level is tested and corrected). This leads to the total number of tests applied being an arithmetic progression. So for r+1 levels of decomposition, the number of tests applied (N_t) is:

$$N_t = \frac{r\,(r+1)}{2} \tag{20}$$

Hence if each of these tests is independent, the total fault detection obtained for this process will be:

$$D = 1 - \prod_{k=1}^{N_t} (1-d_{jk}) \tag{21}$$

Fig. 6 illustrates the average detection per test stage required to achieve a final system quality equal to that of the first level design with this system integration strategy. This detection (d) is a function of the expansion ratio per level (n) and the number of levels (r+1), assuming that d and n are constant per stage of test and expansion, respectively.

Considering the previously given example and assuming equal detection at each stage, the required stage detection can be calculated.

$$\text{number of test stages } N_t = \frac{4(4+1)}{2} = 10 \tag{22}$$

required detection per stage

$$d = 1 - \left(\frac{0{\cdot}105}{1080}\right)^{1/10} = 0{\cdot}603 \tag{23}$$

This is considerably less detection per stage than if tests were applied only during recomposition.

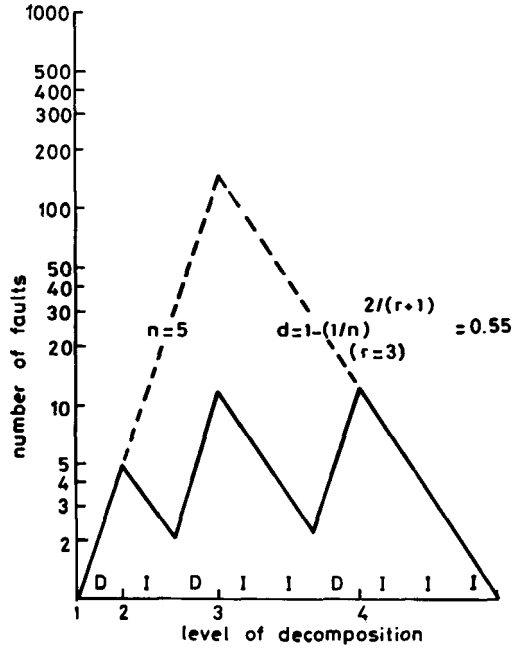


Fig. 6 System integration at each level of decomposition
D: design I: integrate

12 Utility of the model

Taking the mathematical model on its own and accepting the simplifications, especially the assumption of constant detection rate for test stages, it provides a

very convincing justification for the use of a multilevel design and integration development route. The model would also appear to be directly relevant to the software development process and seems to explain the improvements in software quality experienced by the use of formal inspection techniques.

There should be little difficulty in constructing a modelling system using the ideas described here, which would permit the evaluation of the impact of several different testing and development strategies on particular fault classes, providing a database of fault rates and test detections could be constructed. This data could be based, at least in part, on analysis of previous development experiences supplemented by judgments of the impact of change. There is some evidence[9,10] to indicate that this kind of analysis and modelling is effective for manufacturing and field testing. In any case, the process of attempting to model is usually useful in exposing problem areas.

A significant worry in the application of results is the creation of sufficiently independent test methods. It is usually fairly difficult to obtain a large set of systematically designed yet independent test data for a given design. On the other hand, the process of attempting to test, for example assembling modules, is often a test method in itself, since it leads to the discovery of certain kinds of problems. Test independence should not be an insurmountable problem and at least the use of this kind of modelling system provides a set of decomposed tangible targets for the various test stage.

Once a quality plan for development has been established using this kind of technique, performance monitoring is possible, since estimates of performance at various stages have been derived. This is likely to encourage the management to control the production process so that these targets are met. Different techniques may be required to measure the different parameters. For example, actual numbers of faults found may be an adequate indicator of quality, but seeding of faults may be necessary to measure test detection.

One limitation of the model described here is that it does not fully represent the way in which intermodule errors become more likely, due to complexity, as the total size of the design increases. This is an area for further investigation.

Another area for further development of the model is the incorporation of test cost. Increasing the coverage of tests usually results in an exponentially increasing test cost. It would be interesting to investigate the relative merits of increasing the number of test stages against increasing the test cover of few stages from a cost point of view.

Acknowledgment

Acknowledgments are due to Dr. J. Howlett, J.B. Bell, Dr. B.A. Kitchenham and Conway Berners-Lee for commenting on a draft of this paper and for their helpful suggestions.

References

1 GILB, T.: *'Software metrics'*, Winthrop, Cambridge MA, 1977.
2 HALSTEAD, M.H.: *'Elements of software science'*, Elsevier North-Holland, New York, 1977.
3 BOULTON, P.I.P. and KITTLER, M.A.R.: 'Estimating program reliability', *The Computer J.*, **22**, (4), 328-331.
4 McCABE, T.J.: 'A complexity measure', *IEEE Trans.*, 1976, **SE-2**, (4), 308-320.
5 FAULKNER, T.L., BARTLETT, C.W. and SMALL, M.: 'Hardware design faults – a classification and some measurements', 12th IEEE Fault Tolerance Computing Symposium, 1982.
6 MONACHINO, M.: 'Design verification system for large scale LSI designs', *IBM J. Res. Develop.*, 1982, **26**, (1), 89-99.
7 WADSACK, R.L.: 'Fault coverage in digital integrated circuits', *Bell Systems Tech. J.*, 1978, **57**, (5), 1475-1488.
8 WILLIAMS, T.W. and PARKER, K.P.: 'Testing logic networks and designing for testability', *Computer*, Oct. 1979, 9-22.
9 SMALL, M. and BARTLETT, C.W.: 'A Bayesian approach to test modelling', *ICL Tech. J*, 1980, **2**, (2), 207-217.
10 SMALL, M. and MURRAY, D.: 'Evaluating manufacturing test strategies', *ICL Tech. J.*, 1982, **3**, (1), 97-116.
11 KITCHENHAM, B.A.: 'Measures of program complexity', *ICL Tech. J.*, 1981, **2**, (3), 298-316.

Software cost models

B.A. Kitchenham
ICL Mainframe Systems Development Division, Kidsgrove, Staffs.
N.R. Taylor
British Telecom Computer Support Team, Ipswich, Suffolk

Abstract

The paper reports the results of a joint research project by ICL and British Telecom aimed at establishing methods of evaluating software cost models. The document considers some of the problems involved in software cost estimation and some of the features that would be required of a reasonable cost model.

Some methods of evaluating cost models are discussed and the remainder of the document considers applying these principles to the evaluation of two popular cost models. The models evaluated were Putnam's Rayleigh curve model and Boehm's COCOMO model.

The models are evaluated theoretically by considering their underlying assumptions and their internal stability, and empirically by comparing their predictions with historical development data from ICL and BT.

The final conclusion is that neither model can be recommended for use in the ICL/BT environments. It is suggested that the best method of improving cost estimation is to set up a historical database of cost information which can be used initially to assist project managers' judgement and which can provide the data necessary to develop cost models tailored to specific environments.

1 Introduction

Software cost models have arisen because of the need to produce large software systems. Small systems produced by small autonomous teams of analysts/programmers can be expected to be produced more or less to a schedule determined by the individual teams, but experience has shown that large projects usually experience cost-overruns and delays to such an extent that many systems have never been completed at all.[1,2]

Software cost models attempt to provide a coherent model of software development relating the cost of producing a particular software product to the resources available during the various stages of the production process.

Boehm[3] of Thompson Ramo Woolridge (TRW) considers the importance of software cost estimation is that it brings the concepts of economic analysis to the particular world of software engineering, and that it provides an essential part of the framework for good software management. Wolverton,[4] however,

admits that efficient costing methods have been sought by TRW because their customers began to expect the problems of cost and schedule overruns to be born by the developer. Thus, whether we desire better procedures for their own sake or whether harsh necessity forces them on us, it is clear that software cost estimating is an important aspect of large system development.

This paper describes the research undertaken jointly by British Telecom and ICL to investigate two particular software cost estimating models. One model, developed by Lawrence Putnam[5], is Quantitative Software Management (QSM), which is based on describing the relationship between software life-cycle time-scales and the manpower available to produce software in terms of a Rayleigh curve. The other model was developed by Barry Boehm[3] and is based on an algorithmic relationship between product size and production effort refined by the application of a number of cost drivers.

This paper first considers methods of cost estimating in general and then indicates how models may be evaluated.

The Boehm and Putnam models are described in more detail. Both models are then evaluated theoretically and empirically using historical data collected within ICL and British Telecom.

2 Methods of cost estimation

A number of different suggestions have been made for classifying cost estimation techniques.[3,6] Ignoring guesswork, Parkinson's Law and price to win (since such techniques are methods of costing but not methods of cost estimation), the main estimating techniques are considered to be:

(i) *Cost models:* these provide one or more formulas which produce a software cost estimate as a function of one or more variables.
(ii) *Analogy:* this method involves reasoning by analogy with one or more completed projects. The similarities and differences between the new project and completed projects are used to estimate the cost of the new project.
(iii) *Expert judgement:* this method involves consulting one or more experts. When a group is involved, techniques such as the Delphi[7] method can be used to obtain an overall consensus.
(iv) *Top down:* an overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.
(v) *Bottom up:* a software project is split up into its individual components. Each component is separately costed, preferably by the individuals responsible for implementing the component. The individual estimates are then summed to give an overall cost.

Although this is one way of separating the various methods of estimating it is

important to realise that all the methods share certain similarities. The major similarity is that they all require some historical knowledge of other software projects. This is true not only in the obvious case of costing by analogy, but also in the case of even the most theoretical cost models, which must be validated against real data and calibrated for particular environments.

No particular cost estimation technique can obviate the need for systematic and accessible information regarding the cost and nature of completed software projects, and all techniques would be substantially improved by such information. Thus, a *fundamental requirement* for all cost estimation techniques is a historical database recording software cost and project information.

Another important factor is that no one method is consistently better than the others from all aspects, a summary of the relative strengths and weaknesses of the methods based on Boehm[3] is given in Table 1. Boehm points out that the particular strengths and weaknesses are often complementary so that the best method of estimation is a combination of techniques. Thus, a top down estimate using the judgement of several experts based on analogy can be compared with a bottom up estimation using a cost model whose inputs are provided by the future implementors of the systems, in a continuing iterative process.

3 Validation of models

3.1

The first step in validating a model must be to determine the validity of the functional form of the model and its associated parameters. This can be done in two ways:

(*a*) If the model is derived from underlying assumptions about the nature of software development, it may be possible to evaluate the reasonableness of the assumptions.

(*b*) If not, the form of the model and the values of its parameters may be investigated empirically using historical data. The degree to which the model may be validated will therefore depend on the number of parameters and the size of the database.

3.2

The next step is to check the fidelity of the model. This involves two considerations:

(*a*) What is the accuracy of the model, given that the input parameters are completely accurate (the model's theoretical fidelity)?

(*b*) What is the accuracy of the model, given input parameters which are themselves estimates (the model's actual fidelity)?

To check the theoretical fidelity of a model whose input parameter is a number of source statements, the actual number of source statements is used as the input parameter and estimates produced are checked against the actual values.

Table 1 Strength and weakness of software cost estimation methods

Method	Strengths	Weaknesses
Cost models	objective, repeatable	inputs are often subjective
	permit sensitivity analysis	exceptional circumstances not considered
	objectively calibrated to experience	calibrated to past not future
Expert judgement	can assess representativeness of past experience	no better than the quality of the experts
	can estimate the effect of new environmental factors and production techniques	subject to biases
	can cope with exceptional circumstances	
Analogy	based on representative experience	past experience may happen not to be representative
Top down	system level focus	less detailed
		less stable[4]
Bottom up	detailed basis	may overlook system-level costs
	more stable due to cancelling-out effect of aggregation	requires more effort
	fosters individual commitment	

To check the actual fidelity of such a model, the original estimated number of source statements is used as the input parameter.

3.3

The stability of a model may be evaluated by manipulation of the model itself, by simulation or by running various test cases through the model. Manipulating the functional form of the model allows any structural weaknesses in the model to be found. Simulation allows an assessment of the effect of the inaccuracy in estimating input parameters on the model's subsequent cost estimates. Running particular test cases permits a more detailed understanding of the results of using the model.

3.4

There still remains the problem of interpreting the results of a validation exercise. Any theoretical model of software cost assumes that the software development

process represented by the model is the ideal method of developing software. Thus, failure to predict the actual costs of past developments produced without using the model under consideration could be interpreted to imply that the method of past software development was at fault rather than the model.

Similarly, software developments which were costed and managed using a particular cost model may demonstrate a good agreement between estimates and actuals because the development was constrained to follow the model.

4 The COCOMO models

Boehm[3] has developed a hierarchical series of three models under the generic term COCOMO (COnstructive COst MOdel) based on his experience of TRW models and expert opinion. They are called basic, intermediate and detailed COCOMO.

Basic COCOMO is intended to provide quick, early, rough order of magnitude estimates suitable for first cut costing exercises.

Intermediate COCOMO includes additional factors relating to the particular project and its personnel and environment and allows costing to take place at a component level. This additional information is intended to increase the accuracy of the estimate. Intermediate COCOMO is suitable for use once some details of the internal structure of the project have been identified.

Detailed COCOMO, as its name implies, is the most finely tuned version of the model. It attempts to address some of the problems and over-simplifications of the intermediate model by allowing the effects of cost drivers to vary from phase to phase and permitting some cost drivers to influence the estimate at a module level, some at a component level and some at a system level.

4.1 Description of the COCOMO models

4.1.1 Basic equations: All the COCOMO models assume that two basic cost relationships hold, one between the size of the software being developed and the total development effort, the other between schedule (elapsed time) and development effort. The relationships are of the form:

$$MM = a\,(KDSI)^b \tag{1}$$

where MM is development effort in man months, $KDSI$ is thousands of delivered source statements and a and b are parameters dependent on the version of the model being used (basic or intermediate) and the mode of development, see Table 2.

$$TDEV = c\,(MM)^d \tag{2}$$

where *TDEV* is the development schedule in months and *c* and *d* are parameters dependent on the mode of development, see Table 3.

Table 2 Multiplier and exponent terms for the COCOMO effort/size equations

	Model			
	Basic		Intermediate	
Mode	Multiplier *a*	Exponent *b*	Multiplier *a*	Exponent *b*
Organic	2·4	1·05	3·2	1·05
Semidetached	3·0	1·12	3·0	1·12
Embedded	3·6	1·20	2·8	1·20

Table 3 Multiplier and exponent terms for the COCOMO schedule/effort equations

Mode	Multiplier *c*	Exponent *d*
Organic	2·5	0·38
Semidetached	2·5	0·35
Embedded	2·5	0·32

4.1.2 Mode of development: Boehm distinguishes three modes which identify the type of development environment used by software production groups.

(i) Organic mode, which refers to a situation where relatively small software teams develop software in a highly familiar inhouse environment.
(ii) Embedded mode, which refers to the environment established to develop products which operate in a strongly inter-related complex of hardware, software, regulations and operational procedures.
(iii) Semidetached mode, which is intermediate between embedded and organic.

4.1.3 Cost drivers: An important part of the intermediate and detailed models is the concept of *cost drivers*, additional factors which influence the effort required to produce a software product. Boehm identifies 15 cost drivers grouped into four categories.

Product attributes
- required software reliability
- database size
- product complexity

Computer attributes
- execution time constraint

- main storage constraint
- virtual machine volatility
- computer turnaround time

Personnel attributes
- analyst capability
- applications experience
- programmer capability
- virtual machine experience
- programming language experience

Project attributes
- modern programming practices
- use of software tools
- required development schedule

Each is ranked on a scale indicating its importance to a particular product. The ranking determines a multiplying factor which estimates the effect of the attribute on the software development effort. The multipliers are applied to the estimate of effort obtained from the COCOMO effort equation (eqn. 1) to produce a refined estimate of effort.

The three COCOMO models treat the cost drivers differently. In the basic model they are ignored completely, in the intermediate model they are applied either to the whole product or to components of the product. Detailed COCOMO, however, considers some cost drivers at a system level, some at a component level and some at a module level, and further applies different multiplication factors for the cost drivers for each phase of the development process. Thus, for intermediate COCOMO, the multiplying factor for applications experience if it is rated very high is 0·82; for detailed COCOMO it is 0·75 for the product design phase, 0·80 for detailed design and 0·85 for the code and unit testing and integration and test phases.

4.1.4 Phase and activity distribution: In addition to equations for effort and schedule, Boehm provides tabular breakdowns of the effort and schedule distribution for the main phases of software development. This relates the proportion of effort and schedule to phase on a basis of mode of development and size of development.

In addition, Boehm considers the main activities of the software development process and the proportion of time spent on these activities in each phase and produces tables that give details of the percentage of effort involved for each activity, within each phase of development based on the three development modes and the size of the software development.

4.1.5 Other features of the models: The models do not explicitly cost software maintenance but Boehm provides a method of estimating the yearly costs.

In addition, the basic equations assume that the code is to be produced from new, but Boehm provides equations to cater for enhancement or adaptation of existing code.

4.2 Basic assumptions used in the model

It is assumed that:

(i) Eqns. 1 and 2 are valid and that the parameters take the values shown in Tables 2 and 3.
(ii) The different modes of development exist and can be identified.
(iii) The 15 cost drivers selected by Boehm are complete and sufficient and their assigned multiplicative values are valid.
(iv) The proportion of total effort can be split accurately between the various phases and activities of the development process.

5 Putnam's model

Putnam's model is based on Norden's investigations into life-cycle patterns for software projects.[11] The development is considered to be a set of unsolved problems (the problem space) and work progresses until the set is exhausted. Solving problems is assumed to be a sequential process in time such that the occurrence of solutions may be modelled by use of the Poisson model, with associated exponential interevent intervals.

Norden suggested that the manpower curves associated with an engineering development programme could be described using a Rayleigh curve of the form:

$$\dot{y} = 2Kate^{-at^2}$$

where $\dot{y}$ is the instantaneous staffing level, K is the total life-cycle effort and a is a shape parameter (this is Putnam's notation).

Putnam used the basic Rayleigh curve in conjunction with a number of empirically derived assumptions to obtain the equation:

$$S_S = C_K K^{1/3} t_d^{4/3} \tag{3}$$

where S_S is the number of source statements in the final product, t_d is the time at which $\dot{y}$ (the manpower curve) reaches a maximum and is identified with the development time and C_K is the 'technology factor', a constant (for a particular development) subsuming a number of terms measuring:

(i) the state of technology being applied
(ii) the environment in which the development is undertaken
(iii) the development equipment available and the time needed for debugging and testing

(iv) the extent to which modern programming practices are utilised.

Hence Putnam's model postulates a relationship between product size and the development time and total effort for a particular project. The model can be used to show the effects and limitations of 'tradeoffs' between development time and effort (synonymous with cost).

Putnam's model then describes the software lifecycle of the project as a Rayleigh curve, composed of a number of subcycles corresponding to design and code, test, maintenance etc., activities of the project.

Fig. 1 illustrates the Rayleigh curves for $\dot{y}(t)$, the total manpower loading – the project curve – and for $\dot{y}_1(t)$, the loading for the design and coding cycle. The total effort is the area under the project curve and this, from the form chosen for the function $\dot{y}(t)$, is K:

$$\int_0^\infty \dot{y}(t)\,dt = K$$

We define the development effort E as the total effort expended up to the time t_d defined above:

$$E = \int_0^{t_d} \dot{y}(t)\,dt$$

and the design and code effort as the effort expended in this cycle up to the same time t_d:

$$K_1 = \int_0^{t_d} \dot{y}_1(t)\,dt$$

Putnam makes the *assumption* – based on practical experience – that t_d is the time at which 95 per cent of the design and coding work has been completed, i.e.:

$$\int_0^{t_d} \dot{y}_1(t)\,dt = 0{\cdot}95 \int_0^\infty \dot{y}_1(t)\,dt$$

and that this is the stage at which the system becomes operational. He assumes also that the design and coding curve – already assumed to have the Rayleigh form – has the same initial slope as the total project curve, i.e.:

$$d\dot{y}_1(t)/dt = d\dot{y}(t)/dt \text{ at } t = 0$$

It follows from these assumptions and from the mathematics of the Rayleigh form that if t_0 is the time at which the design and code effort is maximum (the

time corresponding to t_d for the total project effort) then

$$t_0 \approx \frac{t_d}{\sqrt{6}}$$

and the design and cost effort K_1 is

$$K_1 \approx K/6$$

Given that S can be estimated, C_K can be assigned a value, eqn. 3 (referred to as the 'software equation') contains two unknowns (K and t_d, the parameters that the model is used to estimate). To derive the software equation from the basic Rayleigh curve, Putnam identified a number of empirical relationships. First he identified the difficulty, D, as:

$$D = \frac{K}{t_d^2}$$

and proposed that the larger the value of D, the harder the system. The difficulty gradient ∇D (often referred to as C) is defined:

$$\nabla D = \frac{K}{t_d^3}$$

By plotting K, t_d and D using existing data, to obtain a difficulty surface, Putnam suggested that the quantity ∇D takes on only discrete values such that, for example:

(i) If the system is entirely new and has many interfaces and interactions with other systems, $C \approx 8$.
(ii) If the system is a rebuild or composite built up from already existing systems, and where much of the logic and code already exist, $C \approx 27$.

Putnam defines productivity PR to be:

$$PR = \frac{\text{total end product code}}{\text{total effort to product code}}$$

As can be seen from Fig. 1, the total effort to produce the code is a burdened number; it includes overhead (and also test and validation) effort, and PR must be adjusted to refer only to the design and code effort.

The adjustment factor is

$$\frac{\int_0^{t_d} \dot{y}\, dt}{\int_0^{t_d} \dot{y}_1\, dt} = 2 \cdot 49$$

Putnam shows that empirical results suggest that productivity could be related to the difficulty by an equation of the form:

$$PR = C_n D^{-2/3}$$

where C_n is a factor related to C_K. From this relationship the software equation 3 is derived.

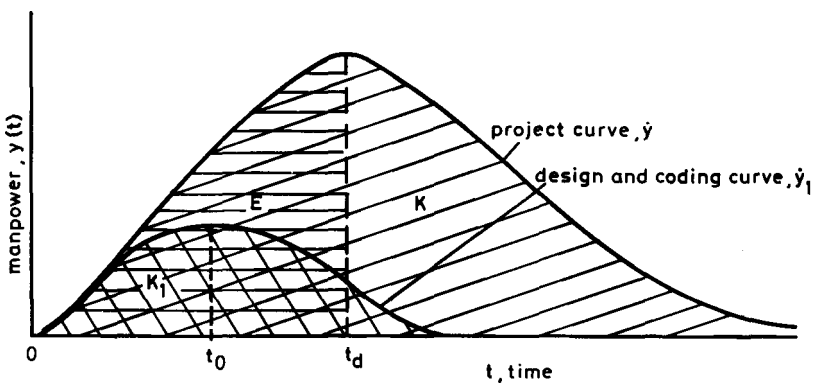


Fig. 1 A comparison of the project curve and design and coding curve

5.1 Summary of assuumptions used in the model

It is assumed that:

(i) Norden's derivation of the manpower curve is valid, the derivation being dependent on the following assumptions:

  (*a*) the development can be thought of as a set of unsolved problems
  (*b*) modelling of solution occurences can be represented by the Poisson model
  (*c*) manpower at any given time is proportional to the number of problems available for solution at that time
  (*d*) there is a linear learning curve associated with the problem solving activity.

(ii) At t=0, $\dot{y}$=0. This means that it is assumed that the effect of the initial specification and requirements analysis in the development curve does not influence the subsequent software development manpower curve. This could be considered to contradict Norden's assumptions.

(iii) The term K/t_d^2 is a measure of the difficulty of a project.

(iv) If a difficulty surface is plotted, using the difficulty (K/t_d^2), total effort

(K) and development time (t_d), then systems will lie on a number of discrete lines, on a surface of the form

$$K/t_d^3 = C$$

(v) The design and coding effort, like the total project effort, is described by a Rayleigh curve; and this has the same initial slope as the project curve.

(vi) The time t_d at which the total manpower loading is greatest coincides with that at which 95 per cent of the design and coding work has been completed, at which stage the system becomes operational.

It follows from points (v) and (vi) that the design and coding loading is at a maximum at a time t_0 such that $t_0/t_d = \sqrt{6}$, approximately; and the effort expended on design coding, the development effort, at that stage is K_1, where $K_1/K = 1/6$, approximately.

Table 4 Constant and exponent terms found by other researchers for effort/size equations

Source	Constant a	Exponent b
RADC	4·55	0·96
Watson and Felix[9]	5·2	0·91
Basili and Freburger[10]		
(i) total lines	1·38	0·93
(ii) developed lines	1·58	0·99

Table 5 Constant and exponent terms found by other researchers for schedule/effort equations

Source	Constant c	Exponent d
RADC	2·58	0·36
Watson and Felix	4·1	0·36
Basili and Freburger		
(i) total lines	4·55	0·26
(ii) developed lines	4·62	0·28

6 Evaluation of the models

6.1 Theoretical considerations

6.1.1 COCOMO:

(i) The COCOMO models are not derived from any underlying mathematical assumptions, but assume that relationships between effort and size, and effort and schedule exist and take a specific functional form:

$$\text{effort} = a \times \text{size}^b$$

and

$$\text{schedule} = c \times (\text{effort})^d$$

The actual values of the multipliers and exponents used by Boehm are shown in Tables 2 and 3.

This form of relationship has been established in a number of other studies as shown in Tables 4 and 5. The figures for the RADC source are quoted in the Putnam tutorial on software cost estimating.[13]

In general, the exponent terms observed by other investigators are fairly similar to COCOMO exponents, whereas the multiplier terms vary quite substantially.

(ii) The underlying assumptions of the model do imply some theoretical problems:

(*a*) The parameters of the model, i.e. the values of the multiplier and exponential in the basic equations, the quantifier associated with each rank of each cost driver and the distribution of effort and schedule across phase and activity, were all estimated by expert opinion with the associated risk of bias.

(*b*) The number of different parameters, when the cost drive quantifiers and the effort, schedule and activity distributions are recognised as such, is extremely large:

For basic COCOMO the following parameters are assumed to be known:

- multiplier and exponential terms in the effort and schedule equations for 3 modes (i.e. 12 terms)
- 70 independent values to determine the distribution of effort across each phase for five given product sizes for each of the three modes
- 42 independent values to determine the distribution of schedule across each phase for five given product sizes for each of the three modes
- 308 independent values to determine the distribution of product activity for each phase for five different product sizes for each of the three modes.

For intermediate COCOMO all the above are required plus:

- 54 effort multipliers corresponding to the non-nominal ratings of the various cost drivers.

For detailed COCOMO all the same requirements as basic COCOMO plus:

- 216 effort multipliers corresponding to the non-nominal ratings

of the various cost drivers in each phase of the development process.

This number of values makes it extremely difficult to validate all the parameters in the model even if a database large enough to cover all the possible combinations were available.

(*c*) There is a dichotomy in the model between the value of the parameters used for the basic COCOMO effort equation, and the intermediate and detailed effort equations.

6.1.2 Putnam's model: As was explained in Section 5, Putnam proposes two equations to summarise software development. The primary equation which he calls the software equation relates code size (S_S) to total life-cycle effort (K) and development time (t_d).

$$S_S = C_K K^{1/3} t_d^{4/3} \tag{3}$$

where C_K is a state of technology 'constant'.

The second equation relates lifetime effort to development time via the difficulty gradient:

$$K/t_d^3 = C \tag{4}$$

where C is a 'constant' related to the type of development being undertaken.

In addition, Putnam relates development effort E to life-cycle effort via the equation

$$K = E/B \tag{5}$$

where B is a 'constant' depending on the size of the development.

The values taken by the three 'constants' are shown in Tables 6, 7 and 8 for C_K, C and B, respectively. In Table 6 C_K is related to a technology factor TF which is sometimes used by Putnam rather than C_K.

Therefore, rewriting eqn. 3 in terms of development effort gives

$$S_S = C_K (E/B)^{1/3} t_d^{4/3} \tag{6}$$

and rewriting eqn. 4 gives

$$E/B = Ct_d^3 \tag{7}$$

The implications of these equations are that:

(i) there is not just one software equation relating size, effort and development times but 138 equations depending on the different values of C_K and B

(ii) eqns. 6 and 7 may be combined to eliminate in turn either t_d or E to produce equations comparable to those used by other researchers and by COCOMO

(iii) eqn. 7 can be adjusted to man months instead of man years and should then be directly comparable to equations investigated by other researchers.

Thus eqn. 7 becomes

$$t_d(\text{MM}) = 12^{2/3} (BC)^{-1/3} E(\text{MM})^{1/3} \quad (8)$$

Table 6 Values of the state of the technology constant C_K and technology factor TF

C_K	610		987		1597		2584		4181		6765	
TF	0	1	2	3	4	5	6	7	8	9	10	11
C_K		754		1220		1974		3194		5168		8362

C_K	10946		17711		28657		46369		75025		121393	
TF	12	13	14	15	16	17	18	19	20	21	22	
C_K		13530		21892		35422		57314		92736		

Table 7 Values of the difficulty gradient constant C

Type of system	Gradient value
New	7·3
Standalone	14·7
Rebuild	26·9
Composite 1	55·0
Composite 2	89·0

Table 8 Values of the size adjustment constant B

System size (in 1000 lines)	Value of B
5-15	0·16
20	0·18
30	0·28
40	0·34
50	0·37
70	0·39
>100	0·39

The permissible values of the 'constant' relating development schedule to effort are shown in Table 9. It seems logical that as the type of development becomes simpler (i.e. C increases in value) for the same size development, the constant should become smaller, but it is not obvious why the constant should decrease as the size of the development increases.

The actual values of the constant of proportionality may be compared with the COCOMO values in Table 3 and the values found by other workers given in Table 5. The nearest equivalent to the COCOMO constant occurs for rebuild systems of 40 000 lines of code. The exponent term corresponds to the value for the COCOMO embedded mode.

Eliminating t_d from eqns. 7 and 8 and converting to man months and 1000 lines of code gives

$$E(\text{MM}) = 12 \times (1000)^{9/7} C^{4/7} BC_K^{-9/7} S_S(KDSI)^{9/7}$$
$$= 86\,533 \frac{C^{0.571}}{C_K^{1.286}} B \quad S_S(KDSI)^{1.286} \tag{9}$$

Eqn. 9 provides a set of 690 different equations relating effort and size depending on the values assigned to C, C_K and B. A subset of the possible constant values is shown in Table 10.

The variation of the constant shown in Table 10 is somewhat paradoxical. It seems logical that the constant should decrease for developments of the same size and same difficulty as the state of technology increases, but it is not at all clear why the constant should increase for developments of the same size with the same state of technology as the type of development becomes easier.

The actual values of the constant and exponent in eqn. 9 may be compared with the values used in the COCOMO models shown in Table 2 and the empirical results obtained by other researchers shown in Table 4. It can be seen that the exponent term is quite high compared with the other proposed values and is closest to the COCOMO embedded mode exponent. The constant term used in

Table 9 Constant of proportionality relating development time and effort

	Values of C				
Values of B	7·3	14·7	26·9	55·0	89·9
0·16	4·98	3·94	3·22	2·54	2·16
0·18	4·78	3·79	3·10	2·44	2·08
0·28	4·13	3·27	2·67	2·11	1·79
0·34	3·87	3·07	2·51	1·97	1·68
0·37	3·76	2·98	2·44	1·92	1·64
0·39	3·70	2·93	2·39	1·89	1·61

Table 10 Constant of proportionality relating size and effort

			C		
	7·3	14·7	26·9	55·0	89·0
$B = 0{\cdot}16$					
C_K					
1220	4·63	6·90	9·74	14·66	19·29
5168	0·72	1·08	1·52	2·29	3·01
10946	0·27	0·41	0·58	0·87	1·15
75025	0·023	0·035	0·049	0·073	0·097
$B = 0{\cdot}28$					
C_K					
1220	8·09	12·07	17·05	25·55	33·76
5168	1·26	1·89	2·66	4·01	4·27
10946	0·48	0·72	1·02	1·53	2·01
75025	0·041	0·060	0·085	0·13	0·17
$B = 0{\cdot}39$					
C_K					
1220	11·27	16·82	23·70	35·72	47·02
5168	1·76	2·63	3·71	5·58	7·35
10946	0·67	1·00	1·41	2·13	2·80
75025	0·056	0·084	0·12	0·18	0·23

Putnam's work seems rather low compared with other proposed values unless the technology constant is assumed to be very low or the software being developed is assumed to be very simple.

However, for both the size/effort and schedule/effort it is clear that Putnam's model implies a functional relationship similar to that proposed by Boehm and observed by other investigators.

There are other criticisms which have been made relating to the basic assumptions of Putnam's model:

(i) Parr[8] has criticised two of the underlying assumptions. He points out that the linear learning curve proposed by Norden is not theoretically supported, and that the skill available depends on the resources, so there is a confusion between the intrinsic constraints on the rate at which software can be developed and management's economically governed choice about how to respond to those constraints.

He then criticises Putnam's assumption that the work done on requirements definition and specification prior to the start of code and design can be ignored. He argues that there are dependencies between problems such that one task cannot begin until others have been completed. Thus, the unsolved problem space is partially ordered. At any point in time a subset of problems exist, called the visible set, which are capable of being worked on. It is the visible set that determines the level of staffing.

Thus Parr concludes that after the requirements have been defined and specified, the visible set of problems has increased so that the staffing levels for code and design need not (and probably *should* not) start at zero.

(ii) Basili and Beane[12] compared Parr's manpower utilisation curve, Putnam's curve, a parabola and a trapezoid with seven actual project records. They found that Parr's curve fitted the data best while Putnam's Rayleigh curve was a worse fit than either the parabola or the trapezoid curves. It is worth noting that five of the seven plots of actual data appeared to confirm Parr's suggestion that the manpower utilisation curve would not pass through the origin at the start of the time period.

6.2 Stability of the models

To investigate the stability of the two models we investigated the estimates they gave when costing a development of 14 000 lines required to be produced in approximately 10 months elapsed time.

For Putnam's model the following values were estimated:

(i) The effort E for a range of technology constants from $C_K = 4181$ (technology factor 8) to $C_K = 17\,711$ (technology factor 14) which is the range of values Putnam reports finding over a wide range of different companies.[15] This effort was calculated for $t_d = 0{\cdot}83$ years (10 months) and $t_d = 1{\cdot}2$ years with $S_S = 14\,000$.
(ii) E was also calculated for $S_S = 12\,000$ and $S_S = 16\,000$ for each value of C_K to investigate the effect of uncertainty in the size estimate.

For COCOMO, the following values were estimated.

(i) The effort for each development mode required to produce 12 000, 14 000 and 16 000 lines of code.
(ii) The schedule for each development mode required to produce 12 000, 14 000 and 16 000 lines of code.
(iii) For semidetached mode, the effort and schedule were estimated
  (*a*) with all cost drivers nominal
  (*b*) assuming analysis capability high, with all other cost drivers nominal.

The results of this analysis for Putnam's model are shown in Figs 2 and 3. Fig. 2 shows the estimated effort for each of the three estimates of code size assuming the development is constrained to take 0·83 years. Fig. 3 shows the estimated effort when the schedule is stretched to 1·23 years. (It should be noted that the scales are different for the two Figs.)

It is clear from these Figs. that there is a strong interaction between the various sources of inaccuracy in Putnam's model. Thus, if the number of lines of code is

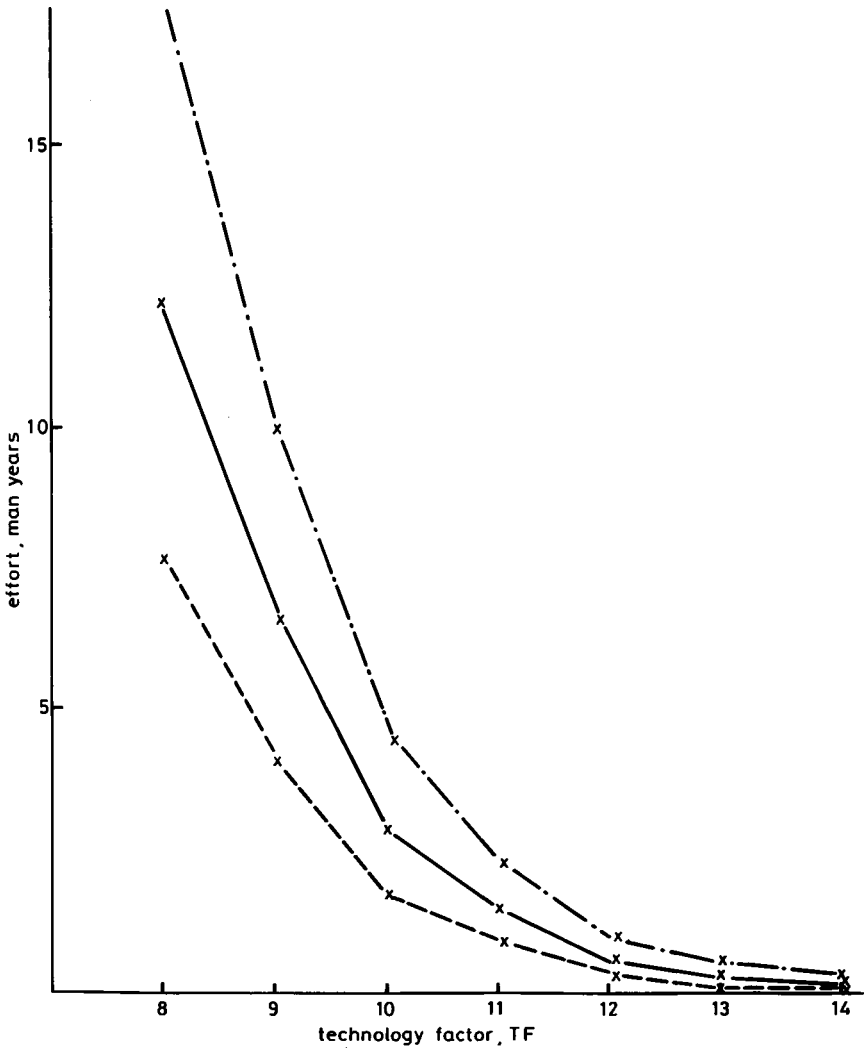


Fig. 2 Estimated effort to produce approximately 14 000 lines of code in 0·83 years using Putnam's technique

underestimated, this will cause a much greater potential inaccuracy in the effort estimate if the technology factor is at the lower end of the range than if it is at the upper end of the range. Similarly, an inaccuracy in the estimation of the technology factor will cause greater inaccuracy if the true value of the technology factor is low.

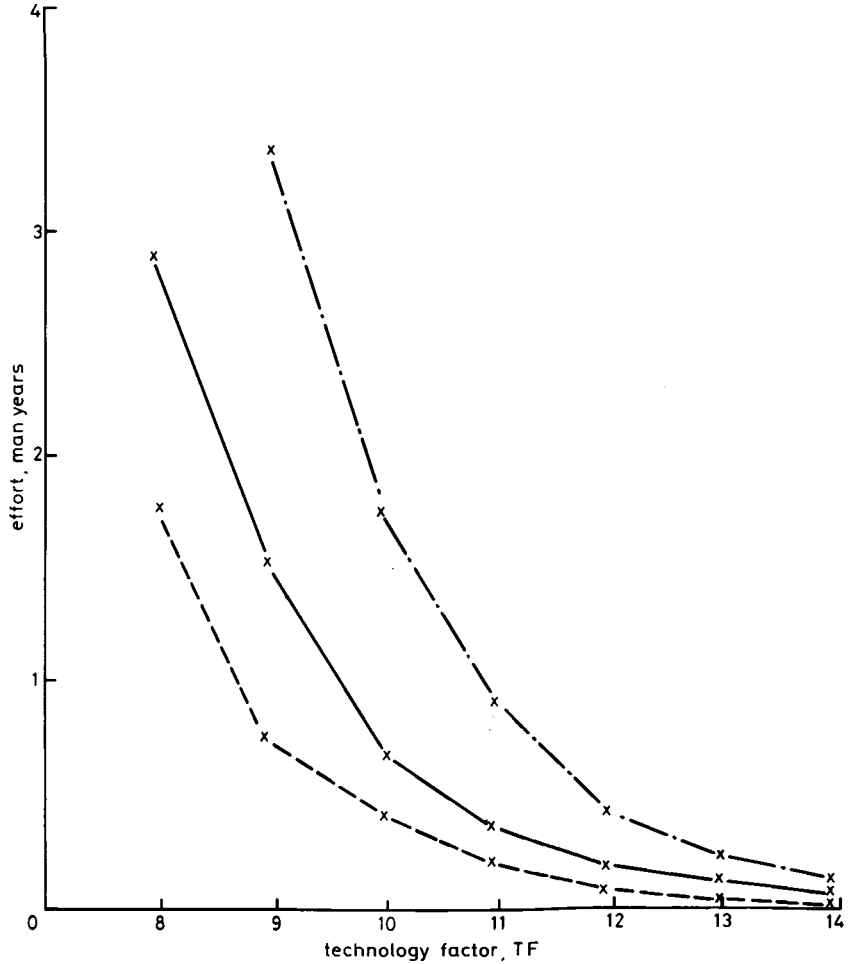


Fig. 3 Estimated effort to produce approximately 14 000 lines in 1·2 years using Putnam's technique

size = 16 000 —·—·—·—
size = 14 000 ————
size = 12 000 - - - - - -

The dependency on the development time is in line with Putnam's predictions and shows that a difference of 0·37 years (approximately 3 months) can have a difference in effort of up to 9·5 man years depending on the technology factor. However, it is again true that the differences in estimates are much greater the smaller the technology factor.

The results of the analysis for COCOMO is shown in Tables 11, 12 and 13. The results are given in man years rather than man months to permit easier compari-

Table 11 Effort in man-years to produce code using basic COCOMO

Mode	Size 12k	14k	16k
Organic	2·7	3·2	3·7
Semidetached	4·0	4·8	5·6
Embedded	5·9	7·1	8·4

Table 12 Schedule in years using basic COCOMO

Mode	Size 12k	14k	16k
Organic	0·78	0·83	0·88
Semidetached	0·81	0·86	0·91
Embedded	0·82	0·86	0·91

Table 13 Effort and schedule estimates, using semidetached mode intermediate COCOMO with (i) all cost drivers nominal and (ii) one non-nominal cost driver

	Size 12k	14k	16k
(i) Effort	4·0	4·80	5·60
Schedule	0·81	0·86	0·91
(ii) Effort	3·40	4·10	4·80
Schedule	0·77	0·82	0·86

son with Putnam's estimates. It appears from these results that the COCOMO estimates are much less variable than Putnam's. For the COCOMO effort equation it appears to be more dangerous to identify development mode wrongly than to estimate size wrongly. The COCOMO schedule equation is very stable but in contrast to the effort equation it is slightly worse to estimate size wrongly than it is to identify development mode wrongly.

However, the effect of altering one of the cost drivers by one level has an effect on the predictions comparable to the effect of either misestimating size or incorrectly identifying development mode, and in the intermediate model there are 15 cost drivers which may be assigned to any one of four or five different levels. Thus, although the basic COCOMO results may be very stable, the effect of cost driver misestimation on intermediate COCOMO estimates could be very serious.

A much more detailed stability/sensitivity analysis will be provided in the final report of this project.[14]

6.3 Empirical results obtained from BT and ICL data

6.3.1 The constant terms of Putnam's model: As part of this study a data collection activity was initiated in both environments, resulting in a joint data-

base of some 20 projects. The BT data was all from projects concerned with the development of original control and operation procedures of advanced telephone switching centres. Many were of a real-time nature, and the projects were undertaken at many development sites. The ICL data came from projects concerned with the development of an operating system and associated file-handling routines, all of which were completed on one site.

Table 14 Estimates of C and C_K for ICL and BT developments

	Project	S	t	K	$C=K/t_d^3$	C_K	TF
	1	17431	1·325	29·70	12·8	3868	8
	2	14142	0·90	57·60	79·0	4214	8
	3	6534	0·75	18·75	44·4	3609	7
	4	3040	0·80	2·08	4·1	3207	7
	5	4371	0·48	12·92	116·8	4956	9
BT	6	15091	2·27	155·42	13·3	941	2
	7	29570	1·00	13·99	14·0	12272	12
	8	23300	0·75	68·94	163·4	8339	11
	9	3000	0·50	7·50	60·0	3862	8
	10	25751	1·90	34·49	5·0	3362	7
	11	19637	1·58	118·89	30·1	2170	5
	12	6050	0·38	3·83	70·0	14049	13
	13	8363	0·51	7·46	56·2	10159	12
	14	13334	0·60	13·16	60·9	11159	12
	15	5942	0·47	1·187	11·4	15355	13
ICL	16	3315	0·26	3·914	222·4	12676	13
	17	38988	1·00	9·673	9·7	18298	14
	18	38614	0·73	1·749	4·5	48757	18
	19	12762	0·50	6·294	8·1	17418	14
	20	13351	0·35	3·556	82·9	35462	17

Table 14 shows the values of C and C_K obtained from ICL and BT data. It should be noted that Putnam's development effort equation (obtained by replacing K by E/B in the software equation) relates to 95 per cent of the detailed design and coding effort plus the test and integration effort which occurred during the time needed to achieve 95 per cent of the detailed design and coding effort. High-level system design and system-feasibility effort are excluded. Thus, where a breakdown of the effort for each stage of development was available the effort and schedule data were adjusted to correspond to Putnam's definition (this was possible for all ICL data), otherwise the gross effort and schedule data were used.

Using the gross effort and schedule data could result in a substantial underestimation of technology factor. However, all the BT results (numbers 1-11) would be subject to this bias, and so comparisons between these projects could be made.

It is clear from Table 14 that the actual values of C are not always close to the theoretical values. Putnam now tends to interpret the difficulty gradient in terms of manpower build-up rate (1983, personal communication) so that any particular type of development – new, standalone, composite etc. – may be

developed in its characteristic manner, resulting in an actual value close to the theoretical C value; or may be developed in a manner similar to a different type of development. This allows the development to take advantage of the slow build up, long development time, low total effort effect.

This avoids the problem that certain types of developments appear to have anomalous C values but adds a further difficulty for anyone who wishes to use the model in a predictive sense as the user needs to know not only what sort of system is being developed but also how it is going to be developed.

Table 14 also shows the actual technology constant C_K obtained for each of the developments. The results show a fair degree of variability even for the ICL developments (numbers 12–20) which were all developed in a similar environment using similar techniques, by in some cases the same development teams (i.e. developments 12 and 19, developments 13 and 14, and developments 17 and 20). This is a particular problem if Putnam's model is to be used predictively because the accuracy of any estimates depends critically upon the accuracy of the estimate of the technology constant (see Section 6.2).

6.3.2 COCOMO estimates: Figs. 4 and 5 compare COCOMO effort predictions for the basic and intermediate models with the actual effort for the BT/ICL

Table 15 Actuals compared with estimates using basic and intermediate COCOMO

		Effort (man months)			Schedule (months)		
		Actual	Basic COCOMO	Intermediate COCOMO	Actual	Basic COCOMO	Intermediate COCOMO
	1	16·7	17·3	16·2	23·0	6·8	6·6
	2	22·6	24·9	21·6	15·5	7·7	7·3
	3	32·2	42·2	17·5	14·0	9·2	6·8
	4	3·9	17·0	17·4	9·2	6·7	6·8
ICL	5	17·3	8·8	10·3	13·5	5·4	5·7
	6	67·7	139·8	160·8	24·5	14·1	14·8
	7	10·1	138·3	102·7	15·2	14·0	12·6
	8	19·3	40·0	34·0	14·7	9·1	8·6
	9	10·6	42·8	16·9	7·7	9·3	6·7
	10	60·5	73·7	98·7	15·9	11·26	12·47
	11	110·5	58·3	56·0	10·8	10·4	10·2
	12	36·0	24·6	NAY	9·0	7·7	NAY
	13 R*	4·0	15·4	NAY	9·6	6·5	NAY
BT	14 R	24·8	20·6	NAY	5·75	7·2	NAY
	15	298·4	62·7	73·4	27·2	10·6	11·2
	16	47·0	133·2	142·5	12·0	13·9	14·2
	17	148·9	102·0	NAY	9·0	12·6	NAY
	18 Firm**	14·4	10·3	NAY	6·0	5·6	NAY
	19	115·9	114·1	196·4	22·8	13·1	15·9
	20 R	256·8	87·0	67·1	19·0	11·9	10·9

* R implies simple rebuild (NAY = not available yet)
** Firm implies firmware

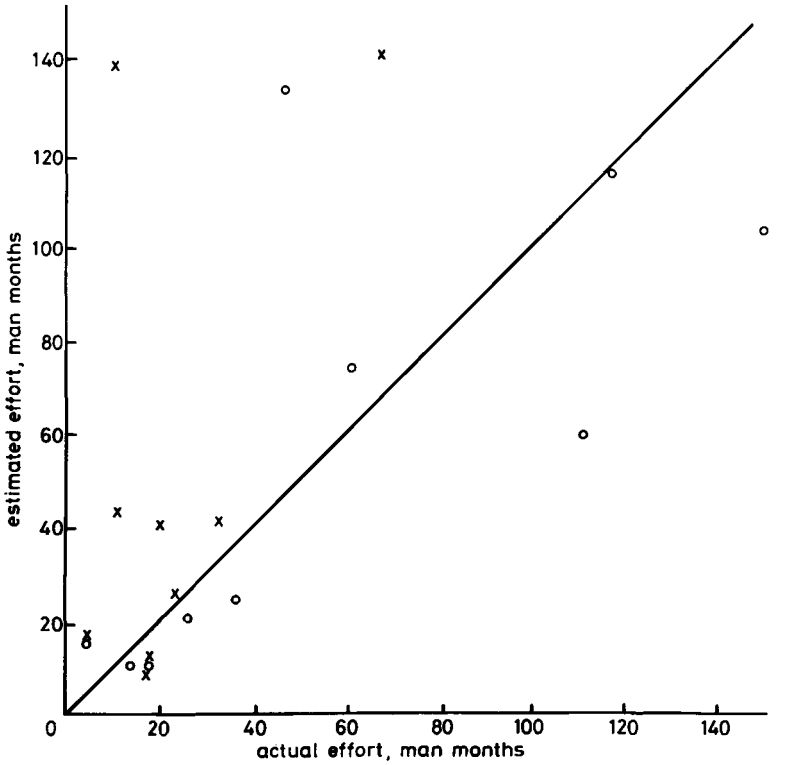


Fig. 4 Relationship between actuals and estimates for basic COCOMO
excluding projects 15 and 20
X ICL
⊙ BT

data. This information, together with the schedule predictions is shown in Table 15. It can be seen that the basic COCOMO predictions match the actual data only very crudely and a few estimates are very poor indeed. The basic model is not intended to give any more than a rough first estimate of costs; however, it is apparent that the intermediate model does not provide any substantial improvement on the basic estimates.

As well as providing gross estimates of effort and schedule COCOMO also permits a breakdown of effort and schedule across phase. The actual breakdown of effort was available for eight out of the nine ICL projects, and is shown in Table 16 together with the COCOMO predictions for developments of size 2K, 8K and 32K lines of code (K = 1000). It can be seen that the average actual values are not dissimilar to the COCOMO predictions. However, it is also clear that the individual values vary widely among developments. Thus, any effort breakdown based on the COCOMO figures could be wildly inaccurate for a particular development, although it could be argued that the COCOMO breakdown might be what should be aimed for.

Table 16 Percentage breakdown of effort by phase

Development size		Product design	Detailed design and code	Integration and test
1	6050	23	32	45
2	8363	12	59	29
3	13334	11	83	6
4	5942	21	62	17
5	3315	11	44	45
6	38988	28	44	28
7	38614	21	74	5
8	13351	7	66	27
Average		17	58	25
COCOMO	2000	16	68	16
	8000	16	65	19
	32000	16	62	22

It is interesting to note that two ICL developments appear to have used a disproportionate amount of effort during the integration and test phase; this would indicate that there were problems associated with these developments. A disadvantage of Putnam's model is that it ignores time spent in integration and test so it would fail to flag these developments as anomalous whereas a cost model such as COCOMO does highlight the problem.

Table 17 Summary of empirical relationships

Relationship		Statistically significant ($p<0.05$)	Per cent of variation accounted for
BT	$EFF = 3.15\ (size)^{1.20}$	√	61%
	$SCH = 3.964\ (EFF)^{0.27}$	√	41%
	$SCH = 4.5\ (size)^{0.40}$	√	39%
ICL	$EFF = 8.196\ (size)^{0.30}$	X	-
	$SCH = 5.6723\ (EFF)^{0.33}$	√	41%
	$SCH = 11.22\ (size)^{0.10}$	X	-
ALL	$EFF = 4.41\ (size)^{0.83}$	√	27%
	$SCH = 7.11\ (EFF)^{0.21}$	√	17%
	$SCH = 6.51\ (size)^{0.28}$	√	22%

where EFF is total effort in man months
size is thousand lines of code
SCH is elapsed time in months.

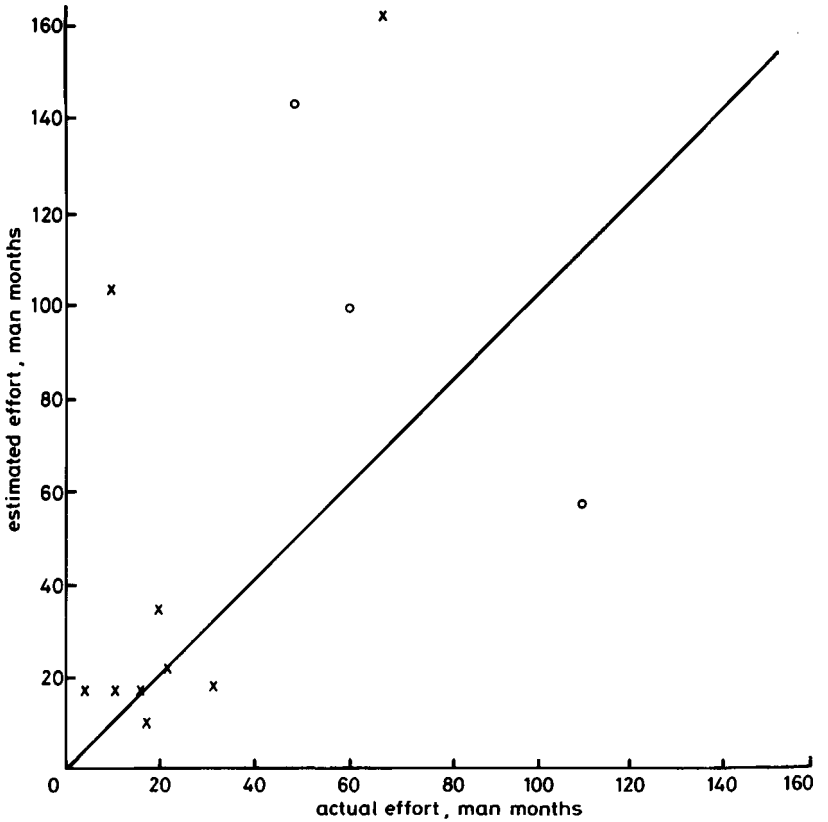


Fig. 5 Relationship between actuals and estimates for intermediate COCOMO

projects 15, 19 and 20 not plotted
projects 12-14 and 17-18 not available
X ICL
⊙ BT

6.3.3 Empirical relationships: Figs 6, 7 and 8 show logarithmic plots of the relationships between size, effort and schedule on the BT/ICL data. Regression analysis was applied to the BT and ICL data using all the data and also on the two sets of data independently. The relations were tested to see whether they were significant over the data points used for their calibration: that is, whether or not the points indicated when there was a definite relation between the variables. A test indicating the percentage of the variation of the y-variable accounted for by the relation was performed also; this can be thought of as a measure of the goodness of fit of the line to the data, 100 per cent indicating a perfect fit. The results of the analysis are shown in Table 17.

The results show a clear difference between BT and ICL which supports both Putnam and Boehm's ideas, as the results could be attributed either to different

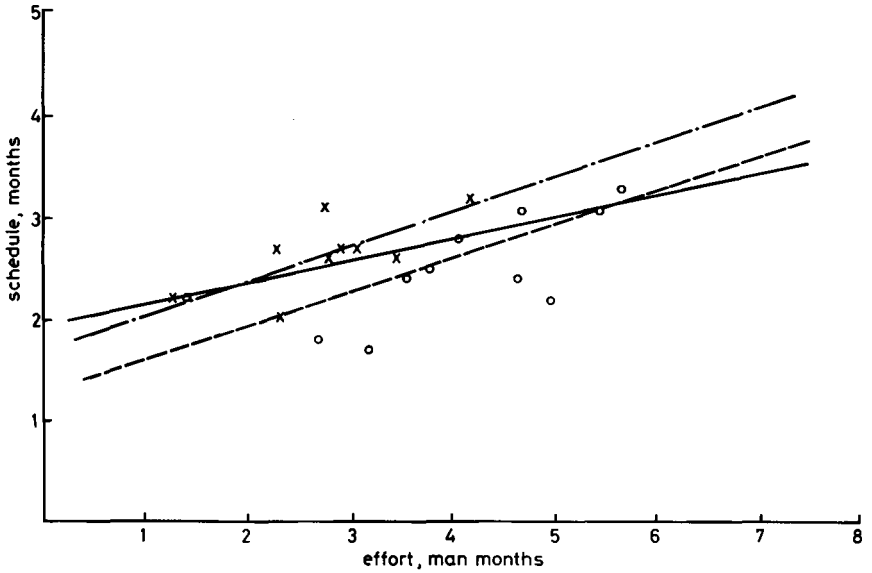


Fig. 6 Relationship between effort and schedule

⊙ BT
X ICL
composite
ICL results
BT results

technology factors or to different development modes between the two environments. The results do emphasise the need to calibrate any models to the particular environment in which it is to be used.

The estimates of the regression parameters shown in Table 17 show some similarity to the COCOMO equations for the effort/size and schedule/effort equations for BT and the effort/schedule equation for ICL.

7 Conclusions

In general, neither of the models performed well in the BT and ICL data sets. Putnam's model was difficult to use because the data to be included and that to be excluded did not conform to the data normally recorded by projects. Also, Putnam suggests his model should be used for medium to large projects, which restricts its applicability, particularly in the ICL environment where small teams produce many relatively small developments. In addition, the exclusion of integration test costs gives spuriously good productivity and technology factors for projects which suffered from integration and test problems which a full development cycle would highlight.

The COCOMO model did not achieve anything like the level of accuracy it

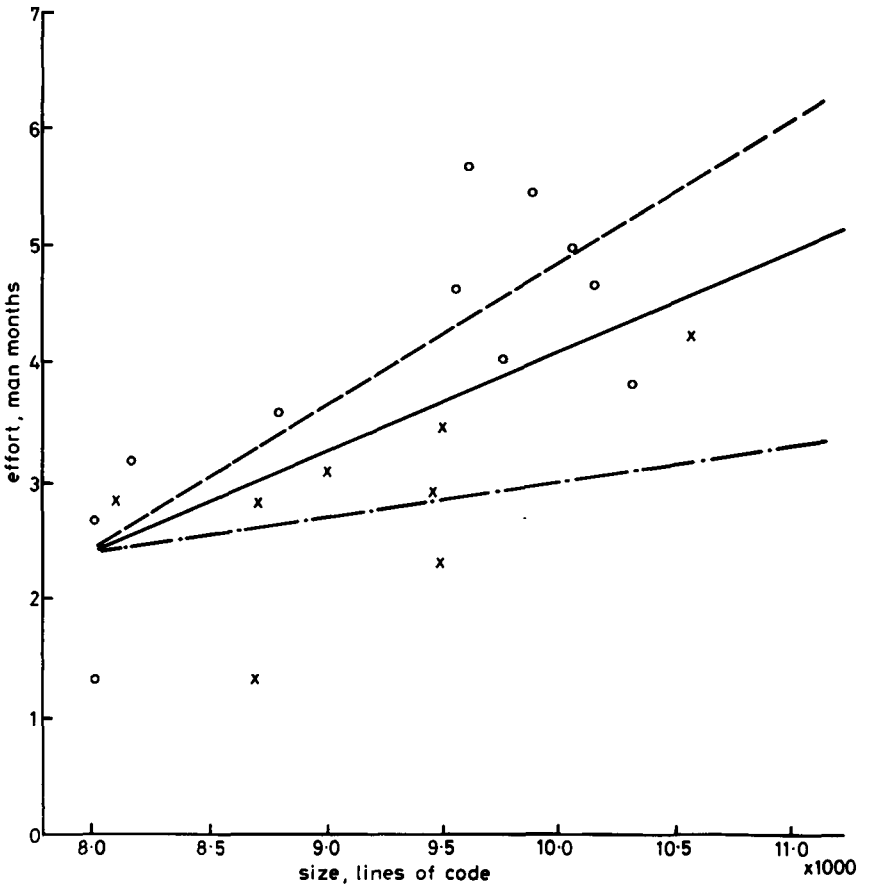


Fig. 7 Relationship between effort and size

attains for Boehm's data set. However, the methodology of the COCOMO model is suitable for developing similar models calibrated to particular environments, which is the procedure Boehm recommends (1983, personal communication). However a fairly large historical database is required – COCOMO was first developed with a database of about 20 developments and needed 60 projects before it was suitably defined. We are, therefore, unable to recommend either model for use directly in BT or ICL environments. To improve cost estimation, our main recommendation would be to establish a database of historical information within each environment which can be used immediately to assist expert opinion and costing by analogy. Such a database may then be used to establish empirical relationships (such as those identified in this document) to provide individual models calibrated to their intended use environment.

For current cost estimation problems we would suggest that empirical relation-

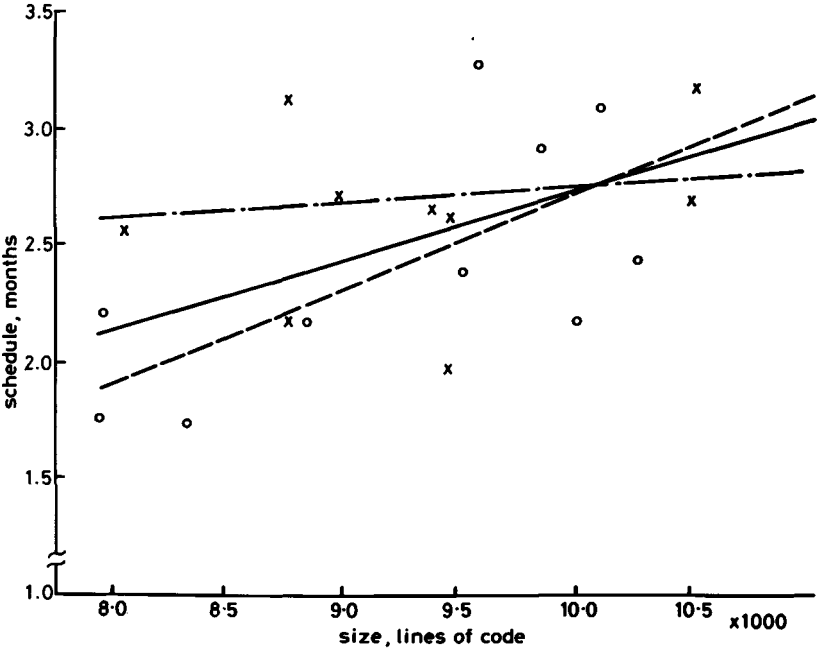


Fig. 8 Relationship between size and schedule

ships identified in this paper can provide a gross check on estimates and that the data set obtained provides the first input to any future historical database.

Acknowledgements

The authors are particularly grateful to Paul Reid for invaluable work on the BT data, and to Christine Lodge for her help in preparing many drafts of this paper. We wish to acknowledge also the permission of the Chief Executive, British Telecom Switching Development Department, to publish the paper.

References

1 BROOKS, F.P.: 'The mythical man month', Addison-Wesley, 1975.
2 BABER, R.L.: 'Software reflected', North-Holland, 1982.
3 BOEHM, B.W.: 'Software engineering economics', Prentice-Hall, 1981.
4 WOLVERTON, R.W.: 'The cost of developing large-scale software', *IEEE Trans. Comput.*, 1974.
5 PUTNAM, L.L.: 'A general empirical solution to the macro software sizing and estimatine problem', *IEE Trans.*, 1978, **SE-4**, *4*.
6 BASILI, V.R.: 'Models and metrics for software management and engineering', IEEE Computer Society Press, 1980.
7 BERNSTEIN, C.B. and CETRON, M.J.: 'SEER: A delphic approach to applied information processing', *Technol. forecast.*, 1969, 33-54.
8 PARR, F.N.: An alternative to the Rayleigh curve model for software development effort, *IEEE Trans.*, **SE-8** 1980, 291-296.
9 WALSTON, C.E. and FELIX, C.P.: 'A method of programming measurement and estimation , *IBM Sys. J.*, 1977, **16**.

10 BASILI, V.R. and FREBURGER, K.: Programming measurement and estimation in the software engineering laboratory, *J. Syst. & Software*, 1981, **2**, 47-57.
11 NORDEN, P.V.: Useful tools for project management, in *Operations research and development*, DEAN, B.V. (Ed.) J. Wiley and Sons, 1963.
12 BASILI, R. and BEANE, J.: 'Can the Parr curve help with manpower distribution and resource estimation problems? *J. Syst. & Software*, 1981, **2**, 59-69.
13 PUTNAM, L.H.: 'Software cost estimating and life-cycle control: getting the software numbers', IEEE Computer Society Press, 1980.
14 KITCHENHAM, B.A. and TAYLOR, N.R.: 'Software cost estimation techniques', ICL/BT Report (in preparation).
15 PUTNAM, L.H.: 'Seminar on software cost estimating', London, 1982.

Program history records: a system of software data collection and analysis

B.A. Kitchenham
ICL Software System Development Centre, Kidsgrove, Staffs

Abstract

This paper describes a semi-automatic system of software data collection and analysis which has been in use in VME production projects since March 1981. The paper outlines the nature of the system itself and indicates the way information about the software development task has been used in practice. It concludes by indicating some of the problems that have been encountered in attempting to use the system.

1 Introduction

Many industrial and academic research workers have stressed the need for and the importance of measurement of software and the software development process.

Some investigators[1,2] have concentrated on the relationship between software characteristics and production constraints to develop cost estimation models; others have considered the relationship between error types and software development techniques[3-5]. However, they have all concluded that the availability of software data is essential for visibility and control of the software development process, evaluation of software products and development techniques, and estimation and prediction of software attributes (e.g. quality or cost).

Nonetheless, there are problems involved with collecting and analysing software data. Basili and Weiss[6] point out that large quantities of data will be generated which will almost certainly require correction and will need to be kept for relatively long time periods (i.e. covering the development and life of a software product). Thayer *et al.*[3] stress the potential cost and schedule impact and lack of resources available for data analysis.

This paper describes a semi-automatic data collection and analysis system used by production staff producing ICL's VME operating system to record details of all enhancements to the VME system. The system is currently only available for SDL developments using the ICL CADES system[7]; however, CADES is being extended to allow projects using any high-level language to use its version control and project librarian facilities, with optional use of CADES database facilities,

and the new CADES facilities will also incorporate the data collection and analysis system.

The system has been in use since March 1981, so this paper not only outlines the system's functions and how it is used but also includes some examples of the practical use of the software data which has been collected.

2 The data collection and analysis system

2.1 Overview

The data collection and analysis system revolves around the concept of program history records (HRs) which provide the mechanism for recording the details of all changes to a software product during its initial development and subsequent maintenance and enhancement.

The main advantage of the HR system is that it is not form-based, it is computer-based. The history records are structured into the form of normal Algol-type macro/procedure calls, which are incorporated into the program which is being developed.

Currently the HR macro calls are put within comment symbols at the start of the program; when the new CADES development is completed, they will exist as self-standing macro calls in the header section of the program. The detailed format of the HR macro is described in a later Section.

Software tools are available to analyse the HR information. One suite of programs identifies the HR macro calls in the amended programs and executes the HR macro; this causes the data incorporated within the macro call to be subjected to preliminary validation and then stored in a normal file which can be edited directly to correct errors in the data. A second suite of programs provides a series of standard summary reports and analyses.

The separation of the extraction and report functions means that the information is available for any additional analyses in a format compatible with other analysis packages (such as the VME 2900 statistics package[8]).

2.2 Format of the HR macro

The HR macro has 15 parameters which may be used to describe a change or enhancement to code, although most of the parameters may be defaulted. A full specification of the HR macro is given in an internal ICL report[9], but a summary of the parameters is given below.

DATE	the date of the change
INITIALS	the person who altered the program

REASON	the reason for the change and may take one of the four values N indicating a newly created program E indicating an enhancement to an already existing program B indicating a change to debug the program D indicating a change for documentation purposes
CLASSIFICATION	this can assume one of 12 permitted values and provides additional information for enhancements and error clearance classifying the type of change
LINES	this identifies the number of non-comment lines of source code involved
OCCURRENCE	this can assume one of 24 permitted values. It applies only to error clearance and identifies at what stage in the development process the error occurred
NUMBER	this parameter is used to allow a number of similar HR calls to be grouped together into one – it is always defaulted to 1
TESTING	this can assume one of 11 permitted values and is used for errors found on in-house services or by customers to indicate by what means the error could have been found earlier in the development process
DAF6 SWNOTICE UREGISTER	these three parameters allow any error to be cross-referenced to other error recording schemes which in VME terms are the DAF6 register, software notice register and the usability register
DRENTRY	this permits a change to be linked to an overall software development project. In VME terms this identifies the development register entry
SYSTEM	this parameter is only used for error clearance and identifies the version of the system which introduced the error. N.B. this allows for the cases where errors in old code are found by new development testing to be identified

SEVERITY	this parameter is only used for error clearance. It may take up to six different values indicating the severity of the error in terms of the effect it had (or would have had) on a user service
EXTRA INFORMATION	this parameter is only used for error clearance. It has 19 possible values which allow for a very detailed classification of the error type (e.g. MLLO means missing logic, line of code omitted)

2.3 Analysis tools

2.3.1 Data extraction. A normal SCL macro interface is available which will extract information from the HR calls in either an individual program or in each of the programs within a standard VME library. The interface permits either the use of all HR calls, or the use of HR calls related to a particular time period.

Once the HR calls have been identified, they are executed as macros which cause the information in the call to be given a preliminary validation and then stored in a specified file. The preliminary validation procedure checks the internal consistency of the information and indicates any missing or misleading information with an asterisk in the output file. If the format of an incorrect HR call violates the normal SCL macro constraints, it will be rejected by the data extraction system and will not be validated nor will it appear in the output file.

If the caller of the data extraction program identifies an output file that already exists the system will append the new information to the end of the file.

The format of the output file is described in detail in an internal ICL document[10].

2.3.2 Data analysis. A normal SCL macro interface is available[10] which provides a number of summaries and analyses of the information obtained by the data extraction program. The reports provide the following information:

- for each program, the amount of change and the number and type of errors found, with overall totals
- for each program, the amount of change and the type of enhancements taking place, with overall totals
- for each program with errors, the stage of the development process at which errors were found
- for each program with errors found by customers or on the ICL in-house service, an assessment of how the errors should have been found earlier
- for each program with errors crossreferenced to code repairs and patches, a list of the patches that were cleared
- two-way tables indicating the relationships between:

- type of error and how errors were found
- code age and how errors were found
- code age and error severity
- error severity and how errors were found
- detailed error classification and how errors were found
- detailed error classification and the severity of error

The data analysed may be all the data in a particular output file, or the data relating to a particular time period, or the data relating to a particular DRE.

2.4 Control

To ensure that HR records are maintained by the production staff, the CADES system does not permit the transfer of a new version of a program into trusted filestore unless there is an extra HR call compared with the old version of the program. However, there is no check on the internal consistency of the HR call until the data extraction and analysis programs are used.

3 Use of HR information

Information obtained from the HR macro calls may be used for a number of different purposes including individual project control and quality control, gross productivity figures and assessment of new working methods. In this Section a number of examples will be given of the way HR information has been used.

3.1 Individual project control

The HR analysis programs may be used to give 'snap-shots' of the state of a particular development by indicating the programs that have been involved in the development with the amount of work progressed and error screening patterns to date.

Simple crosschecks between programs scheduled to change and programs which have been changed can (and do) reveal changes which have been overlooked.

3.2 Development quality control

Fig. 1 shows a scatter diagram of program size against number of errors for a small development (N.B. the system does not produce the plot, it provides the data for the plot). This shows a fairly typical pattern of a nearly linear relationship between size and number of errors. Such relationships seem relatively normal for VME developments[5] and indicate that the development has been broken down into individual programs reasonably well. A disproportionate number of errors in either large or small programs would indicate potential problem areas with the overall design.

Another important point to be gained from such scatter diagrams is that specific anomalies are pinpointed. In Fig. 1 the largest program (which is circled) appears

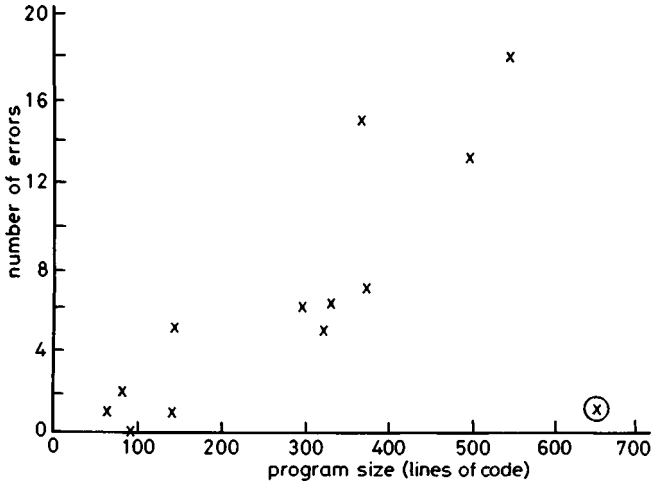


Fig. 1 Relationship between program size and number of errors (development 1)

to have disproportionately few errors. This implies that the program must be checked to ensure that it really is error free and not that it has not been properly tested. The program in fact was a control program which identified what work was required and directed subsequent processing into more complex programs. It was therefore large but fairly simple and its simplicity was the reason for its low error rate.

Fig. 2 shows quite a different development pattern. The scatter diagram is separated into five regions with different characteristics. Region A covers the majority of the programs in the development. It shows a relatively random distribution of errors with the error rate for larger programs quite low. Region B

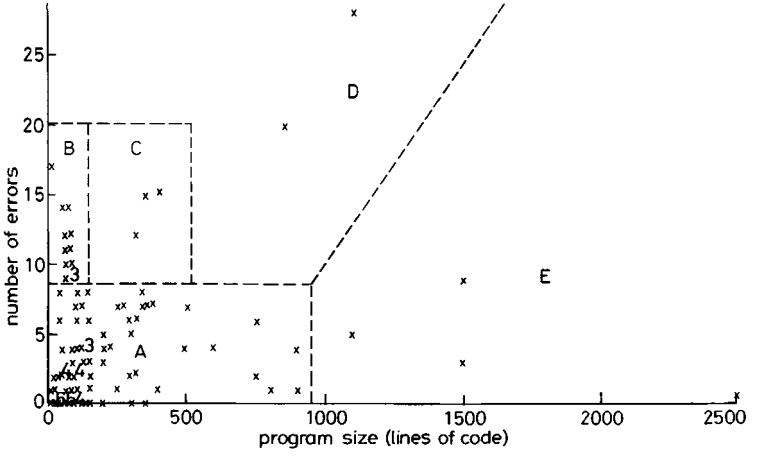


Fig. 2 Relationship between program size and number of errors (development 2)

comprises a number of relatively small programs with relatively high error rates. They were a series of interrupt handlers with similar problems and were highlighted by the team leader as an area for redesign. Region C includes three programs of moderate size and moderately high error rate which were the initialisation and contingency handling routines. These programs exhibited the high error rate often found in programs that interface with other subsystems. Region D includes two programs of moderate size and very high error rates. These programs were written by an inexperienced person and included a large amount of complex parameter checking. The programs were subject to a large amount of testing effort. Region E comprises four very large programs with very low error rates. These programs were transformations of already existing and tested programs so the low error rate was anticipated and confirmed by the results. This development again shows the use that can be made of simple development statistics but emphasises the importance of the detailed knowledge of the project or team leader for interpreting any statistical results.

3.3 Project costing and estimation

The information on size of code produced and the breakdown into new code enhancements to existing programs, unplanned changes and error clearance, can be related to work schedules and development effort statistics to provide historical information to improve future time scales and cost estimates.

3.4 Gross productivity and quality trends

Gross statistics tend to be less interesting than individual developments but they do reveal important trends if the information is available over a reasonably long timescale. Table 2 shows the planned, and unplanned production within the VME production projects between October 1981 and June 1983 contrasted with

Table 1 VME production and error screening between October 1981 and June 1983

Time period	Planned production per month (lines of code)	Unplanned production per month (lines of code)	Error screening per 100 lines
October 1981 to March 1982	15 852	4 554	2·02
April 1982 to August 1982	26 283	5 289	1·65
September 1982 to December 1982	39 114	5 715	1·40
January, 1983 to June 1983	22 411	4 876	2·77

the error screening rates (based only on errors found in-house). This demonstrates an almost inverse relationship between software production and error screening implying that achieving very high productivity figures will reduce the time available for clearing errors.

Tables 2 and 3 show the overall trends for error type and the methods by which errors are found during the development process. These tables can be used as the current 'standards' by which to judge both the progress of individual developments, and the effect of different development techniques.

Table 2 Type of errors found (January 1983 to June 1983)

Type of error	Number of errors	Percentage
Software interface	79	2·1
Design	495	13·3
Code	2613	70·3
Other	528	14·2

Table 3 Method of error screening

Method of error screening	Number of errors	Percentage
Dry checking and code inspection	995	37·6
Testing by code execution (test beds and development base)	1353	51·2
Exposure on in-house services	295	11·2
Customer bug report	215	
Unclassified	1072	

3.5 Assessing working methods

The use of detailed project data to help assess project working methods has demonstrated the need for methods which will improve the detection of design errors, and which will increase the efficiency of current techniques[5]. Practical results outlining the effect of changed working methods have been observed and will be reported in a future paper.

3.6 Empirical tests of software engineering hypotheses

Collection of data from real-life software projects can sometimes furnish the information needed to determine the validity of theoretical models of software engineering.

For example, there are several schools of thought with regard to reliability: one

extreme view is that reliability need not be measured if the software is proved correct. However, it is more usual to accept that current development methods cannot guarantee error-free software, so there is a need to estimate the reliability actually achieved.

There are two quite different techniques used to estimate reliability; one method involves analysis of failures observed when a program executes in its intended environment[12], the other method involves error counting throughout development[13]. Results from VME data show that in one project 155 errors were cleared in code released to customers between January 1983 and June 1983 of which only 80 were found as a result of customer bug reports, the other 75 being found by the production staff and by the in-house services.

This suggests that error counting models would seriously underestimate the reliability of VME as perceived by customers. Thus, an accurate measure of reliability should be based on execution in the anticipated use environment.

4 Problems with the system

The experience gained using the system since March 1981 has shown a number of practical uses of the data as outlined in Section 3, but have also indicated a number of problems.

Practical problems arise because programmers are required to record a large amount of data for each code change for error clearance and there are usually between 1 and 3 errors to clear for each 100 lines of code produced. When time pressures arise there is a tendency to put in the minimum amount of information and not to worry about using the correct keywords or positional parameters. This means that, as Basili and Weiss[6] found, a good deal of time is spent correcting data and it is not always possible to obtain accurate data, since the programming staff forget what sort of errors they were dealing with after a 3-6 month interval.

To help the programming staff to construct their HR calls, it has been found useful to attach a 'crib sheet' (see Appendix) outlining the permissible variables but it probably requires management interest and commitment to maintain the quality of records.

Theoretical problems arise in two ways, firstly because of the nature of the classification scheme and secondly because of the problems of dealing with a continually evolving product. The first problem arises because the current version of the scheme only allows an error to be classified in one way, for example a ripple error cannot also be classified as a design or code error. Thus the classification scheme itself does not preserve mutually exclusive categories and the user of the scheme cannot remedy this defect by multiple classification.

The second problem arises because VME code has a relatively long life expectancy and will be amended and enhanced throughout. Thus, during enhancement of

Appendix Table 4 HR macro checklist which is placed beside videos as an aid for production staff

parameters: keywords and position

1 DATE 2 INITIALS 3 REASON 4 CLASSIFICATION 5 LINES (of SDL) 6 OCCURRENCE (how bug was found) 7 NUMBER (total number of changes recorded default = 1) 8 TESTING (how bug should have been found)

9 DAF6 10 SWNOTICE 11 UREGISTER 12 DRENTRY 13 SYSTEM (where code is) 14 SEVERITY 15 EXTRA INFORMATION

Classification–
enhancements

planned changes :
N – new development

unplanned changes :
E – high-level specification change
I – high-level design change
D – design/specification deficiency
O – other
X – unknown
M – mode change
S – standards

Testing
D/DC – desk checking
U/SCL/PRO – unit/project testing
C – code execution
R – residual error
E – special environment
X – unknown
TVP – TVP testing
FI – inspection

System
null – not released
IDS
ICRS
valid system id
(e.g. 6.11)

Reason
B – bug
E – enhancement
N – new code
D – documentation

Classification – bugs
I – interface to S/W
H – hardware
C – code/logic
D – design
R – 'ripple'
S – source clearance error
L – compilation
E – EP/database
O – other
X – unknown

Occurrence
DC/D – desk checking
SCL/PRO/U – unit/project testing
A – alpha testing
B – beta testing
DB – development base
IDS – IDS query
ICRS – ICRS query
IQE/QE/Q – QE query
BCRS – Bracknell ICRS
OCRS – other in-house
C – customer query
X – unkown
FI – formal inspection
TVPDC/TVPD – TVP dry checking
TVP/TVPSCL/TVPRO/TVPU – TVP

Severity
F – system crash/idle
M – VM crash/idle
L – job failure
T – trivial
X – unknown
H – service severely degraded

Extra information

LCCI	loop count
LCWU	while/until
BCSC	simple branch control
BCCB	compound Boolean
MLLO	line omitted
MLUR	unitialised RC
MLUV	uninitialised variable
MLCO	condition overlooked
S3DE	dereferencing error
S3BI	misuse of BIPS
S3CC	cast/coercion wrong
GESR	slicing wrong
GETD	TLD wrong
GEIA	array index wrong
GEWV	wrong variable
GEWM	wrong mode
IFIC	call set up wrong
IFIR	return values
N.B.	add/R for repeats

existing software old errors may be cleared which may or may not have been found by customers. All these cases can be distinguished by the system but it is fairly cumbersome and requires the DRE parameter and the SYSTEM parameter to be specified correctly in each HR call. This aspect of the system is also difficult to validate automatically so that errors in the HR calls are unlikely to be identified.

5 Final comments

The HR system has been used within the VME production projects since March 1981 and has provided a number of insights into the nature and effectiveness of VME development methods and the quality of individual software developments. It is not possible to make quality statements about the VME system as a whole, however, because the system was not instituted at the start of VME development. It is likely that a completely new software product would benefit even more from such a data collection scheme if it was instituted at the start of development, and the extension of CADES to languages other than SDL/S3 will allow other ICL projects to take advantage of both the CADES facilities and the HR data collection and analysis scheme.

Acknowledgments

The format of the data collection scheme owes a great deal to the helpful and perceptive comments provided by Allen Kitchenham. It was entirely Allen's idea to structure the history records in the format of a macro call and to use that feature to facilitate the data analysis system. I should also like to thank Brian Hayselden for providing the detailed technical expertise necessary to interpret some of the statistical results. In addition, John Fellows did the programming necessary to extend the basic data analysis routines to include the two-way tables needed to investigate relationships between the various data items. It should also be noted that without the co-operation and patience of the VME production staff, there would be no data for me to analyse. Finally, I should like to thank Christine Lodge for her help with the production of the manuscript.

References

1 BOEHM, B.W.: *'Software engineering economics'*, Prentice-Hall, 1981.
2 PUTNAM, L.H.: 'A general empirical solution to the macro software sizing and estimating problem', *IEEE Trans.*, 1978, **SE-4**, 345-361.
3 THAYER, T.A., LIPOW, M., and NELSON, E.C.: *'Software reliability'*, TRW Series of Software Technology 2, North-Holland, 1978.
4 WEISS, D.: 'Evaluating software development by error analysis: the data from the Architecture Research Facility', *J. Syst. & Software*, 1979, **1**, 57-70.
5 KITCHENHAM, B.A.: 'The use of software metrics to assess software production methods', Proc. FTCS-13 Milan, 1982.
6 BASILI, V.R., and WEISS, D.M.: 'A methodology for collecting valid software engineering data', University of Maryland technical report, TR-1235, 1982.
7 McGUFFIN, R.W., ELLISTON, A.E., TRANTOR, B.R., and WESTMACOTT, P.N.: 'CADES – software engineering in practice', *ICL Tech. J.*, 1980, **2**, (1), 13-28.
8 COOPER, B.E.: 'Statistical and related systems', *ICL Tech. J.*, 1979, **1**, (3), 229-246.

9 KITCHENHAM, B.A.: 'The format and use of Holon history records', DRS/PN/2007 3.1, 1982.
10 KITCHENHAM, B.A.: 'Tools for the analysis of SDL Holon history records', OSTC/IN/2233 1, 1982.
11 KITCHENHAM, B.A., and TAYLOR, N.R. 'Software cost models', *ICL Tech. J.*, 1984, **4**, (1), 73-102.
12 LITTLEWOOD, B., and VERRALL, J.L.: 'A Bayesian reliability growth model for computer software', *Appl. Stat.*, 1973, Series C, **22**, 3.
13 REMUS, H., and ZILLES, S. 'A prediction and management of program quality' Proc. 4th Intern. Conf. Software Engineering, 1979.

Notes on the authors

Dr. A. McKerrell Solution of the global element equations on the ICL DAP

Dr. McKerrell graduated from the University of Glasgow in 1965 with a Ph.D. in Theoretical Physics. After appointments at Princeton, Iowa State and Cambridge he was appointed to a Lectureship in the Department of Applied Mathematics (now the Department of Applied Mathematics and Theoretical Physics) at Liverpool in 1968.

Professor L.M. Delves Solution of the global element equations on the ICL DAP

Professor Delves graduated from Oxford with a D.Phil. in Theoretical Physics in 1960. After appointments at the University of New South Wales and at Sussex he was appointed to the Chair of Computational Science (now Computational Mathematics) at Liverpool in 1969.

T.L. Faulkner The Atlas 10 computer (with C.J. Pavelin)
Quality model of system design and integration (with M. Small)

Trevor Faulkner is the manager responsible for product introduction and technical support within ICL's Atlas Division and manages the design authority function for Atlas 10 products within ICL as a whole. He joined ICL from Elliot Automation at the time of the formation of the company in 1968 and worked in Stevenage on the design and development of the 7903 and of an early 2900 processor. Moving to West Gorton in 1975 he managed the development of various 2900 products, including the system integration of 2950, 2955 and enhancement to 2966 before joining the Atlas 10 project in October 1981.

M. Small Quality model of system design and integration

Mike Small graduated from Brunel University in 1967 and has worked for ICL since then. He has been involved in the development of the 1900 and 2900 series and the ME29, his principle contribution to these developments having been to develop test design methodologies. He has published a number of papers on this subject in this Journal and elsewhere. His current post is as a system designer in the Knowledge Engineering Business Centre at West Gorton.

Dr. C.J. Pavelin The Atlas 10 computer (with T.L. Faulkner)

Dr. Pavelin graduated from Cambridge in 1964 and worked for three years in the Computer Division of the English Electric Company (which later became part of ICL) before going to Edinburgh to read for a Ph.D. He joined the Science Research Council's Atlas Computer Laboratory (now the Computing Division, Rutherford Appleton Laboratory, Science & Engineering Research Council) in 1972, where he is Head of the Systems Development Group.

Dr. K.J. Turner Towards better specifications

Ken Turner graduated in Electrical Engineering at Glasgow University in 1970 and went on to study at Edinburgh, where he gained a Ph.D. in 1974 for his work on computer perception. After this he joined ICL and worked on many aspects of communications design and implementation. He is currently running a project which is studying specification and development methods for networking.

Dr. B.A. Kitchenham Program history records: a system of data collection and analysis
Software cost models (with N.R. Taylor)

Barbara Kitchenham received a B.Sc. in Mathematics and Statistics (1969), M.Sc. in Statistics (1970) and Ph.D. (1972) at the University of Leeds, and worked for three years as a statistician before joining ICL as a systems programmer. After several years working on the production of the VME operating system she moved into the area of software management and since 1980 has worked on the problems involved in measuring software and in integrating measurements into the development process. She has published a number of papers on software development, based on empirical studies of VME production.

P.D. Hall O.B.E. The ICL University Research Council

Peter Hall retired from ICL in April 1980. He had been with ICT, where his career had included periods as Main Board Director responsible for the development and production of the ICT medium and large computer systems – 1904 and upwards, Atlas and Orion – and as Marketing Director. On the merger of ICT and English Electric Computers in 1968, to form ICL, he became responsible for Personnel, Education and Training and also for Software Development and Field Engineering. Later he returned to marketing, and retired as Director of Corporate Communication. He was appointed to the University Research Council on its formation in 1982.

Peter Hall has been an active Governor of Queen Mary College, University of London for several years and chairman of the College technology-transfer company QMC Industrial Research Ltd. He was President of the British Computer Society in 1981/2 and Chairman of Council during the Duke of Kent's term as President for the Silver Jubilee year 1982/3. He gave the Faraday Lecture of the Institution of Electrical Engineers in 1966.

N.R. Taylor Software cost models (with B.A. Kitchenham)

After working for a year at the UK Atomic Energy Authority's Fluid & Heat Transfer Laboratory, Neil Taylor graduated in Mathematical Sciences at Teesside Polytechnic in 1982. In that year he joined the British Telecom System X Software Support Division, where he is now working in the Computer Systems Support Group.