ICL TECHNICAL JOURNAL

Volume 5 Issue 3 May 1987

Published by
INTERNATIONAL COMPUTERS LIMITED
at
OXFORD UNIVERSITY PRESS

ICL TECHNICAL JOURNAL

The ICL Technical Journal is published twice a year by International Computers Limited at Oxford University Press.

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

1987 subscription rates: annual subscription £32 UK, £40 rest of world, US $72 N. America; single issues £17 UK, £22 rest of world, US $38 N. America. Orders with remittances should be sent to the Journals Subscriptions Department, Oxford University Press, Walton Street, Oxford OX2 6DP, UK.

Printed by H Charlesworth & Co Ltd, Huddersfield ISSN 0142–1557

ICL TECHNICAL JOURNAL

Contents

Volume 5 Issue 3

The ICL Fifth Generation Programme

Guest Editorial

The Japanese announcement of their "Fifth Generation" co-operative project in the autumn of 1981 was the trigger for a rush of other Fifth Generation programmes to emerge in the USA and Europe. In the USA this has partly taken the form of new co-operative ventures, such as the Microelectronics and Computer Corporation (MCC) at the University of Texas, supported by over 20 firms in the industry. This centre is far larger than the Japanese ICOT research centre, established by MITI using staff seconded from the eight large Japanese firms engaged in the programme. And the MCC is by no means the only US Fifth Generation programme. Of course defence money is helping to speed up the work in the USA, notably through the Defence Advanced Research Project Agencies (DARPA) Strategic Computing Programme – not to be confused with Star Wars.

In Europe the UK's Alvey Programme, and the EEC's ESPRIT programme are the most visible sign of work on the Fifth Generation. But, as in Japan and the USA, all the main computer firms have work going on on aspects of the Fifth Generation, almost by definition of the next generation of computing. And it is often forgotten that the European Computer Industries Joint Research Centre at Munich (ECRC) is a Fifth Generation research centre somewhat larger than ICOT, even if far less visible. This centre is funded by ICL, Bull, and Siemens, and houses a team of high quality. Some of its work is featured in this volume.

If it was the Japanese announcement that triggered the Fifth Generation programmes of the Western World, or at least gave publicity to them, it was the recognition that the applications of Artificial Intelligence will lead to large new markets that has led to their funding by firms and governments. Expert Systems are going to become a feature of every considerable walk of life. And the eventual solution of the difficult problems of general purpose speech recognition and Natural Language understanding is going to unleash a whole new range of applications of computers as it becomes easier to "converse" with one's system.

Since the days when Alan Turing was the unsung prophet on the use of computers for symbolic and logical manipulation, not just for the solution of mathematically based algorithms, the UK has had an honourable place in the vanguard of the development of the techniques of Artificial Intelligence computing that lies at the heart of the Fifth Generation. Edinburgh University and other UK research centres have provided a steady stream of

workers in the field, and have pioneered many of the functional and logical language techniques. ICL with its DAP was one of the first computer manufacturers to pioneer the use of parallel computers, which will be essential if the power required to handle all but the most simple of Knowledge Based Systems is to be available. The significance of the DAP is probably more widely recognised abroad, where it has been flattered by being copied, and it probably remains the parallel machine for which the most software has been written.

So it is nice to see the leading part that ICL has played in both the Alvey and ESPRIT programmes. This volume brings together articles on a whole series of projects in which the company is engaged. Seen together it makes a most striking collection, both for its breadth and for its quality. The way in which ICL have developed their close links with the excellent work of the British Universities is clear from the list of authors. Is there any other firm in Europe who could put together such a record of their work for the Fifth Generation? It must give the Japanese food for thought!

Brian Oakley
Director, The Alvey Programme

Foreword

Every industry evolves by a process of steady development – interspersed with major discontinuities. These occur when technology, industry and customer understanding and the economics of business coincide to cause a jump forward to the next plateau of development.

The so called Fifth Generation potentially represents such a discontinuity in the Information Technology industry and in this issue of the ICL Technical Journal we examine some facets of this against the background of the overview of the ICL programme in this area as outlined in the first paper.

The real and valuable result of any discontinuity will be the impact these technologies have on systems applications. It is in this area that Fifth Generation techniques demonstrate the economic value over alternative systems approaches. The first set of papers describe some current applications using these new techniques.

To achieve this, some radical rethinking will be required in the applications environment – the way the system and the user interact. The set of papers from the European Computer Industries Research Centre (ECRC) describe the basic research goals being explored in this area.

The key technology streams of new language types and the development of more powerful and parallel machine architectures provide the underpinning technical capability and the papers in this area draw heavily on the work carried out under the UK co-operative research programme – Alvey.

We are making no predictions as to when Fifth Generation technology in total will have major user impact, however what is clear is that the ingredients for a substantial step forward in the quality and usability of information systems are now coming together.

J.M. Watson
Director, ICL Marketing and Technical Strategy

What is Fifth Generation? – the scope of the ICL programme

B.J. Procter and C.J. Skelton
Mainframe Systems Division, ICL, West Gorton, Manchester

Abstract

The scope, motivation and modus operandi of the ICL Fifth Generation programme is outlined to provide a context for the remaining papers in this issue of the ICL Journal.

1 Introduction

The single chip microprocessor has brought a computer-based revolution to the 1980s. In human terms, however, the applications we run on these processors are extremely simple. In the 1990s we will take for granted the ability to readily use wide ranging knowledge bases. This knowledge will be accessed from our offices, factories, shops, homes, transport vehicles and used in our business, social and leisure activities. Information will be easy to obtain and it will be possible to get answers which are inferred from rather vague questions. By today's standards the applications required to provide these knowledge-based systems will be complex and they will require very powerful but low cost machines to give acceptable response times. The ICL Fifth Generation programme seeks to provide the software and hardware technology which will bring this new computer based revolution to the 1990s.

2 Fifth Generation

Strictly speaking "Fifth Generation" is the name that the Japanese coined for their ambitious "Fifth Generation Computer System" programme. The name is now in common use but without a commonly accepted definition. It will be used here with the context and scope of the ICL programme.

Fifth Generation (5G), like its predecessor generations, is a technology for developing and delivering user solutions, and consists of a set of techniques, languages, tools and machines. From a practical standpoint it is important to appreciate that many of these components can be used individually. It therefore becomes possible to introduce them incrementally and users can adopt an evolutionary exploitation path to fit their organisational, cultural and configuration needs. The broad structure and relationship of the parts of

the technology is depicted in Fig. 1, and the next few paragraphs provide a very short introduction to each of the main areas and illustrate the scope of the ICL programme. Later papers in this issue treat these subjects in greater depth.

3 Applications

The object of the technology is the construction and execution of applications and it is useful to start by considering just where it can be used or, since the theoretical field of application is very wide, where it can be first applied to best effect. Within the fields of interest of ICL's customers this suggests two broad classes – applications which would be very difficult to produce or maintain using conventional techniques and those which have very heavy processing requirements.

Examples of the former class are applications containing significant amounts of knowledge, those required to make decisions where qualitative or judgemental factors need to be taken into account and those which support deep or adaptive human-computer interaction. These types of application are often termed "AI" (because the techniques are derived from those developed in the field of research known as "artificial intelligence") or simply "intelligent", or "knowledge-based". This class is broadened to a much wider set if there is also a requirement for the application to be subject to a high rate of change – for example the support of operational procedures where the procedures themselves change frequently. Fifth Generation languages encourage a clear separation of data, the rules for interpreting the data and the rules for applying it. This separation makes it much easier to update

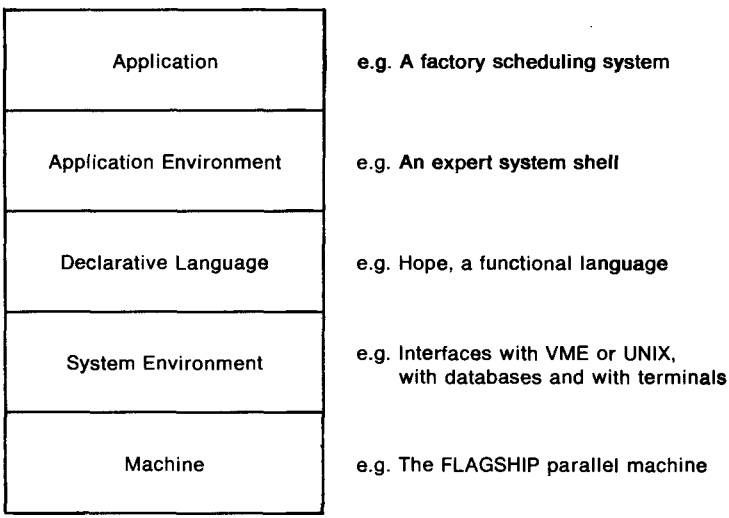


Fig. 1 Components of Fifth Generation technology

applications to accommodate changing requirements. The usual nomenclature used to describe the combinations is:

Data	The set of raw fields (e.g. in a database).
Information	Data + a set of descriptions to define its meaning (i.e. what is being represented and in what form).
Knowledge	Information + a set of rules governing its application.

The other main class of application – with high processing loads – is most frequently encountered by ICL users in database operations, where the processing load is often accompanied by high disc rates. This latter class is expected to grow quickly as more use is made of "fourth generation" tools such as Quickbuild, Querymaster and relational databases to accelerate the production of applications.

4 Application environment

The Application Environment provides the immediate facilities for the development of new applications and for their execution. The facilities may consist of a highly packaged tool which provides a very effective development route for a particular type of application (such as a domain-specific expert system shell), or a more comprehensive set of tools and languages which vastly widen the application scope at the expense of more work in building the application (such as an AI toolkit). Even in this latter case the Application Environments are designed to be highly productive, providing methods and representations chosen for the directnesses with which they can represent the knowledge model of the application. The nearest analogy is with databases and dictionaries, which provide a ready way of representing the data used by an organisation and which allow the use of fourth generation languages to build applications which model the processes built upon the organisation's data-flow. The Application Environment aims to provide the same capabilities for the knowledge within an organisation.

5 Declarative languages

Declarative languages are high level languages in which solutions are described by a set of logical rules. Contrast these languages with traditional imperative programming languages in which the solution is described in terms of a "recipe" which will yield the desired result when run on a computer – in other words, the methodology for solving the problem and the mapping of this solution on to a sequential von Neuman computer have got mixed together.

Several benefits arise from separating these two concerns: programmer concentration is directed to the more important global aspects of the solution, leaving compilers to supply the lower level implementation; the

program strategy is more directly relatable to the problem, simplifying understanding, reducing errors and easing maintenance; because the languages have (mathematically) simple semantics, it is practicable to build powerful transformation and reasoning tools to improve application lifecycle quality and efficiency; and because the programmer has not unnecessarily constrained the implementation, the program can be ported across a range of implementations including parallel machines.

The historical objection of the loss of runtime efficiency which can arise through the use of a high-level language can be offset by compiler and transformation optimisations which are a one-time investment. In fact, because these languages have, to a greater or lesser degree, abstracted away from strictly sequential semantics, they make it possible to exploit the great cost-performance advantage of parallel machines without throwing yet more burden on the application programmer.

6 System environment

Interaction with the outside world, and particularly with existing databases, applications and networks, is vital to allow the fifth generation components to be smoothly integrated into existing systems. The System Environment complements the declarative languages by providing the set of primitives through which all higher layers of software interact with the rest of the system. As with the declarative languages, the aim is to match the functionality of these primitives to the problem and to hide unnecessary implementation detail. This is the point in the system where any temporal constraints essential for the correct solution to the problem are explicitly introduced. By exposing only the essential problem dependencies at this coarse level, the understandability of the solution is preserved and the minimum constraints are placed on parallel execution.

The system environment must deal with all the system management jobs required to run a secure, reliable, multi-user, mixed workload system. For the foreseeable future this task will be accomplished in part by software in the declarative system and in part by the operating system (VME or UNIX) of the host system. The Fifth Generation system is therefore implemented as a sub-system within a classical host environment, where the sub-system consists of 5G software components which execute directly on the host, or in a special purpose (parallel) machine attached to the host. In the case of dedicated workstations the host environment may be invisible to the user.

7 Machines

A parallel machine is one in which a number of identical processing nodes operate on a single job to improve the response time within the job. By contrast, in a present day multiprocessor, the processing nodes are applied to separate jobs within a workload to improve the overall throughput and reduce job queueing. Whereas multiprocessor configurations rarely exceed

four processing nodes, parallel machines can be configured up to hundreds. It therefore becomes possible to choose the most cost-effective technology and architecture for the individual processing nodes and to achieve the desired level of system performance by replication.

There are many different forms of parallel processor suited to differing tasks. Some, like the search engine in CAFS[1], have a specialised architecture dedicated to one particular task whilst others, like the Distributed Array Processor, DAP[2], belong to a class of machine known as "single instruction multiple data". Within the ICL 5G programme, most attention is focussed on "multiple instruction multiple data machines" with globally accessible stores. Their ability to deal with the dynamic problem topologies encountered where operations are data dependent makes them most suitable for declarative languages and for AI applications.

For the present, parallel machines are not freestanding – a conventional system acts as a host and provides the overall operating system and external communications. An existing database will probably already be accessible to the host, but for some applications the parallel machine will generate a very large database traffic requirement. In these cases it may be appropriate to attach parallel disc channels directly to the parallel machine, which then assumes the role of a database or knowledge base machine.

8 The ICL 5G programme

In summary, therefore, the ICL fifth generation programme includes the following components:

- Application development using expert system shells, a modelling tool and an AI toolkit.
- Acquisition and development of an Application Environment.
- Development and porting of declarative languages, compilers and tools.
- Development of a system environment.
- Development of parallel machines.
- Development of object managers which are used to hold information and knowledge bases.

It was not possible to arrive at the fifth generation systems technology by a simple process of evolution. ICL needed to rapidly build up expertise in an interlocking jigsaw of new techniques and to develop new skills. Fortunately, encouraged by earlier SERC programmes, the UK has built up a solid academic base in these techniques. When the ICL University Research Council was formed, one of its major aims was to provide a two-way transfer of technology and techniques between ICL and academia. The contacts and relationships built up through this route have been instrumental in helping ICL to shape and implement its 5G programme.

The UK's response to the Japanese Fifth Generation Programme has been the establishment of the Alvey Programme to promote and help fund

projects under the themes of VLSI, Software Engineering, Intelligent Knowledge Based Systems (IKBS), Human Computer Interaction (HCI), Computer Architectures and a number of Large Scale Demonstrators. An important and fruitful prerequisite of these projects has been collaboration between university research groups and industrial project teams.

Within ICL, the Marketing and Technical Strategy Directorate has taken the lead in setting up a 5G programme through its centrally funded Group Technical Strategy (GTS). A set of interlocking projects has been chosen for its closeness to ICL's needs in the late 80s and early nineties. To date, these projects are mostly joint projects within the Alvey programme. There is space here only to mention those most relevant to the new computer systems architectures, which form the theme of this issue of the journal.

9 The ICL Alvey projects

The DHSS Large Demonstrator, led for ICL by Charlie Portman, aims to provide decision support systems to assist in a variety of tasks arising in large, legislation based organisations. These include:

- The application and maintenance of large volumes of legislative conditions.
- Application of conditions which require interpretation, including the identification of relevant case law.
- The provision of advice to potential claimants both on their eligibility for benefits and how to make a claim.
- The formulation of changes in policy and the exploration of their possible effects.

The DHSS Demonstrator team includes ICL, Logica, the University of Lancaster, the University of Surrey, Imperial College London and the DHSS.

The IPSE 2·5 Project is led by Mike Tordoff and Brian Warboys. It is a joint project between ICL, STC and Manchester University, and seeks to provide an advanced Integrated Project Support Environment of a generic nature, which supports the use of rigorous approaches to mainly software system development based on formal methods and highly integrated management tools.

The High Speed Multi-User Prolog Data-Base Machine project has a team at Heriot-Watt University led by Professor Howard Williams and Professor Fred Heath. The ICL leader is Michael Quinn. It is building a large Prolog database and is applying the ICL CAFS engine to speed database searching.

The Persistent Information Space Architecture (PISA) project is led by Nick Capon and Graham Pratten for STL, with work at the Universities of Glasgow and St. Andrews led by Professor Malcolm Atkinson and Professor Ron Morrison. The team is designing an object-oriented system that allows

data to be treated in a consistent manner regardless of where it is actually stored or who is referring to it.

The DACTL project is spearheaded by a team at the University of East Anglia led by Professor Ronan Sleep and Dr. John Glauert. Imperial College and the University of Manchester are also collaborating and the ICL co-ordinator is Nic Holt. DACTL defines a graph reduction model of computation and an associated language to be used as a portable compiler target standard. DACTL is used as an intermediate language within the FLAGSHIP project.

Parlog is a logic language developed at Imperial College which is very well suited to parallel execution. The Parlog on Parallel Architectures project is developing compilers from Parlog to DACTL and to the High Speed Prolog Database machine. The academic partners are Imperial College, led by Dr. Keith Clark, and the Heriot-Watt team. The ICL leaders are Michael Quinn and Ken Watts. The route via DACTL makes Parlog available for the FLAGSHIP and GRIP machines.

The Graph Reduction in Parallel (GRIP) project is being implemented at University College, London, by a team led by Dr. Simon Peyton-Jones. It is led for ICL by Phil Broughton with High Level Hardware Ltd. as industrial partner, and is a rapid implementation of a graph reduction parallel processing machine, exploiting the powerful 68020 microprocessor on a bus architecture.

Flagship is Alvey's largest project and forms the basis for many of the articles in this issue of the journal. It covers the Hope functional language, program transformation tools, a parallel processing machine hosted by VME and UNIX systems and an operating environment. The consortium is led by Colin Skelton and Brian Procter for ICL and includes teams at Imperial College London, led by Professor John Darlington, Hugh Glaser and Mike Reeve; and at the University of Manchester, led by Dr. Ian Watson, Dr. Viv Woods and Professor Warboys.

10 The ICL motivation

The present 5G programme has been designed to respond to a number of pressures, some recent and some visible within the industry for many years. The main drivers have been:

The high cost and unpredictability of software development and maintenance, coupled with the shortage and mobility of trained programming staff. This has been a constant problem within the industry for two decades in spite of significant advances in software engineering methodology and more recently with the widespread adoption of fourth generation systems. It used to be called the "software crisis", but the word "crisis" seems inappropriate

for a phenomenon of such longevity. "Limit to growth" is a more durable phrase which better captures the effect of this problem.

The advent of the cheap PC has brought much closer the ideal of a computer to each person. It has also thrown into sharp focus the minimal availability of applications for anything other than rather mechanistic or clerical tasks. Quality and productivity aids for managers and professionals are of strategic value to organisations in an increasingly competitive world. Unfortunately they are extremely difficult to build using conventional software technology. Fourth generation systems have helped with some types of application but their stylisation limits their range. More wide-ranging tools are needed to build systems which can adapt and respond to people and to human organisations (rather than the other way around). Such tools are now emerging from research into the commercial world.

Historically, the workhorse of the information technology industry has been the single processor, or at most a multi-processor usually not exceeding four processors. As the user demand for performance (at constant cost) has risen, the computer industry has met these demands by producing faster unit processors, exploiting developments in the semiconductor industry to achieve the speed-up. Whilst there is still active growth and considerable scope for further developments at the lower performance end of both industries, there are signs that the cost of maintaining performance improvements of the very high performance circuits is escalating. With this background, there is worldwide activity to find a convenient way of achieving the required very high performance by harnessing a mutiplicity of processors from the fast-developing cost-effective technologies. The crucial word is "convenient" since it is not generally acceptable to achieve performance at the expense of more complicated programming.

The exciting prospect offered by fifth generation technology arises from the way in which its constituents assist each other to overcome these problems. The new software power tools, needed to build the intelligent applications of the future, provide opportunities for the improved software engineering methodology which will itself be required to maintain quality in such complex applications. A new way of providing the large computing power required by intelligent applications is enabled by the very languages needed to express the applications in the first place. Big improvements in the cost-performance of high-speed computers will allow more extensive computer assistance in the application development process and further development of languages and tools towards the problem domain. The components of the fifth generation can be said to form a true symbiotic relationship.

Whereas the points mentioned above present an argument for innovation, it is recognised as crucial to protect the investment in existing technology of ICL's customers and of ICL itself. This investment is in databases, in applications, in the skills of the data-processing staff, in equipment and in product lines. Major themes running within each project and throughout the

whole 5G programme are integration and evolution. The aim is to allow the technology to emerge from the development labs into products in an incremental fashion, and to allow the increments to be integrated with existing installations and product lines. "Revolution by evolution" is the slogan.

The papers that follow in this issue of the Technical Journal, whilst not aiming at an exhaustive coverage of the programme, will give an indication of the innovative work that is being done.

References

1 The ICL Content Addressable File Store (CAFS), ICL Tech. J. 1985 4(4) 351–506 (12 papers).
2 HOWLETT, J., PARKINSON, D., SYLWESTROWICZ, J.: 'DAP in Action', ICL Tech. J. 1983. 3(3) 330–344.

APPLICATIONS

The emphasis of the entire Fifth Generation programme is on the solution of problems, and the problems that are presenting themselves are becoming more and more complex and increasingly of the knowledge processing type. The papers in this section describe a small number of applications; at this stage in the attack on a problem attention is concentrated on assembling the knowledge on which the solution must be based.

The Alvey DHSS Large Demonstrator Project

E.C.P. Portman
ICL Knowledge Engineering Business Centre, Manchester

Abstract

The paper describes a 5-year collaboration being undertaken by ICL, Logica Ltd., the universities of Surrey and Lancaster, Imperial College London and the Department of Health and Social Security and supported under the Alvey scheme. It is in the field of Intelligent Knowledge Based Systems and its aim is to help members of the general public in making claims under the UK Social Security provisions and staff of the Department in assessing the eligibility of these claims.

Introduction

The Alvey Large Demonstrators are intended to bring state-of-the-art technology in a number of chosen areas to bear on practical problems, in order to demonstrate the ability to provide solutions in previously difficult or even impossible areas. The chosen technological areas are those of Intelligent Knowledge Based Systems (IKBS), Software Engineering (SE), Man-Machine Interface (MMI) and Very Large Scale Integration (VLSI). The DHSS Demonstrator, in its pilot systems for benefit administration in the Department of Health and Social Security (DHSS), makes direct use of all but VLSI of these. It is expected that delivery vehicles for these systems will rely heavily on the continuing rapid fall in hardware costs and, to some extent, on the novel architectures being made practical by the capabilities of VLSI.

The 5-year project, started in April 1984, has been exploring problem areas and the relevance of techniques and is now concentrating on specifying and building exemplar systems for demonstration and evaluation. It is being conducted by a consortium with 35 staff drawn from ICL, Logica, the universities of Surrey and Lancaster, Imperial College London and the DHSS itself; the consortium is investigating five promising representative application areas of the activities of the DHSS, which is seen as an example of a large legislation-based organisation.

Application areas

The areas being studied are chosen for their importance to the DHSS and for the difficulty of applying classical Information Technology (IT) methods to them in a satisfactory way; they concern the following activities:

1 The work of the DHSS Policy Branches, which is in formulating and maintaining the legislation and regulations controlling eligibility for benefits.
2 The claiming of benefits: potential claimants are often confused about the benefits available and about the choice of those applicable to their own case; and can find this a major deterrent to claiming.
3 The process of making an application for benefit: this also is often seen as a major difficulty.
4 Assessing eligibility: the DHSS officers have to apply, in each particular case, the appropriate sub-set of a very large volume of rules and regulations, and in many cases their decisions must be guided by those made in previous cases drawn from a large quantity of historical material.
5 Training: in an organisation as large as the DHSS the training problems are formidable; systems with the capabilities we envisage should be of great benefit to the trainers as well as to their pupils.

Demonstrations

Four demonstration sub-systems are being designed and built to gain practical experience of building on this scale, to provide demonstrations of capability and to allow experimental use in friendly environments, real or simulated. These will be evaluated in various ways, and rebuilt and re-evaluated once more before the project completes in early 1989; they are

- the Policy Advice System
- the Claimant Advice System
- the Forms Helper
- the Local Office System

aimed respectively at the activities of formulation and maintenance of rules and regulations, selection of benefits by the claimant, making an application for benefit and assessment of entitlement.

We shall show how such systems will enable the staff responsible for formulation and upkeep of rules and regulations to be aided in evaluation of alternatives, in consistency checking and in elimination of certain undesirable features such as the "traps" described in the next section. We shall show also that many potential claimants will be able to select benefits appropriate to their circumstances and to make successful claims more easily than is possible at present. Finally we shall be able to support the decision-making of the officers who have to evaluate the claims, by helping them select rules

and cases relevant to the application being considered and by recording the basis for each decision in case of future query. By use of IKBS and MMI techniques we shall be able to develop these systems with relatively small amounts of effort and to make them easy to use and to up-date when changes in regulations have to be made: such systems can be efficiently and reliably built by proper use of SE techniques.

It must be borne in mind that present large DP systems are not only costly to develop and validate but also are almost impossible to change quickly without introducing errors. This makes it very difficult for legislators to act quickly to meet changing circumstances and causes major problems for the DHSS, for example, in introducing changes. It is expected that a stable core of software will exist in the systems that we are proposing and that changes will be incorporated by making changes to the knowledge base: this will be much easier, both for writing and for checking, than changing computer code and also will be amenable to various automatic checks for consistency and completeness.

We see these four systems working as parts of a whole, with much of the knowledge that is used being drawn from common sources; this will improve consistency across the different areas.

Results to date

Listed below are some of the results we have achieved so far; some have led to blind alleys, others are being incorporated in our current and future work.

We have built a number of pilot systems to explore the application of various IKBS techniques to the problems identified in our analysis. These have included various schemes for knowledge representation and for inference; an architectural scheme using standard tokens and having some of the properties of a "blackboard" system; and the use of entity/relation modelling to support a semi-automatic way of generating some of the code and data. These pilots have been assessed for effectiveness, in some cases by further detailed studies. The pilots are as follows.

(i) A pilot in the claim-assessment area, to address the "income" and "requirements" aspects of Supplementary Benefit. This was used also to explore an entity/relation tool for modelling the interfacing to "browsers" and to large DHSS databases.

(ii) One early pilot explored a technique dubbed "gentle evaluation", for allowing changes of mind to be made when filling a form without demanding a complete return to the start. This was seen as important in allowing a more natural interaction with the machine than is common in current expert systems.

(iii) The "Forms Helper", designed and built by the group at Surrey, has been the subject of formal study, with tests made by potential claimants and a comparison of paper and electronic form filling; the

results will be of great value in designing the final Claimant Information System. The Forms Helper attempts to aid clients by guiding them to answer all the relevant questions and steering them away from irrelevant ones on the claims form; it does this by providing explanations and giving examples of expected answers for each box on the form, and in some cases checking that the claimant's answers conform to expectations. The user can go back and change answers if desired.

(iv) An experimental forms toolkit was devised to provide a base on which to build forms interfaces. This allows fields to be expanded if the volume of data entered is too great for the pre-assigned space; and also gives pop-up detail fields for questions with compound answers, like "Your address ..."

(v) The first Policy pilot explored the detection of "traps" in the legislation, for example claimants being prevented from moving to higher levels of benefit as a result of choosing more attractive initial benefits. It explored also the effects of proposed changes in legislation on the classes of claimant endangered by such traps.

(vi) The selection of relevant case history was explored in two different "browser" developments; these differed in their methods but both aimed to make it possible for the user (a DHSS officer) to select relevant sub-sets from a large set of cases, using re-formulations of selection criteria to focus in on a small number of relevant examples.

(vii) One pilot system explored a possible basis for training staff in the application of rules to cases.

We have also developed and explored

- the use of an analysis tool to assist in extracting and classifying material from transcripts of interviews
- an interactive planning tool to assist in managing the project
- numerous representation and inference methods, to support our needs.

In addition we have implemented a method for bi-directional inferencing that not only enables one representation of rules to be used both to deduce consequences from given facts and to seek facts that will support a desired consequence, but also, because it supports explicit representation of bi-conditionals, allows inference of disjunctive and negative conclusions and the use of disjunctive and negated premises. We have integrated this reasoning capability with the ability to handle arithmetic relations, which we believe to be a novel achievement.

In this work we have been exploring both the "frame" techniques used in some early expert systems and also the logic programming concepts used by other researchers and from which the PROLOG programming language was developed. Our present view is that both techniques are relevant and we are defining a toolkit that will make use of both.

The results obtained in these pilot studies, together with other recent results published in the general literature or developed in other Alvey projects, will be used in building our application demonstrations.

Other results

There are other outputs of this project that are as important as the demonstrations to the overall purpose of the Alvey programme of improving the potential competitiveness of the UK.

We are developing an "engineered and measured approach" to building systems of this type that will give others who follow a basis for estimating the time and effort needed for the task and for the development and the delivery of the machines that the systems will need. We are writing a "methodology portfolio" of techniques for each stage in the process, using our own experience in creating these demonstrations to validate each step. In particular we intend to describe analysis techniques for defining the area to be covered and the potential value to the target organisation; elicitation techniques for defining and categorising the various groups of knowledge needed to operate the systems; and the building techniques that are used to define the software environments in which the knowledge will be held and used. We shall describe also the project-management techniques that we have used in this unusual distributed project, which reaches across six organisations, across academia, industry and public administration, across numerous disciplines and physically across seven sites distributed around the UK.

In this work we are growing a group of experienced application designers who will be of great value to any organisation intending to create systems of this type for sale. During the coming years we shall publish papers on the detailed experience of defining these systems, and on our evaluations of their potential value to the DHSS.

Acknowledgements

The work described in this paper was carried out as part of the DHSS Large Demonstrator Project, supported by the Science and Engineering Research Council and the Alvey Directorate of the UK Department of Trade and Industry. The views expressed are those of the author and are not necessarily shared by his collaborators.

PARAMEDICL: a computer-aided medical diagnosis system for parallel architectures

Martyn G. Cutcher
Systems & Architectures
Malcolm J. Rigg
Public Services Business, ICL Industry Systems, Reading, Berkshire

Abstract

The paper describes, in general terms, an expert system loosely modelled on MEDICL (AAP) and written in the parallel logic language PARLOG and intended for running on parallel architecture computers, notably the Flagship machines. This work was carried out as part of the UK Alvey Flagship Project; PARAMEDICL being the proposal made by Public Administration Business Centre (PABC) – now the Public Services Business (PSB) – and chosen by a selection panel from many Applied Systems submissions. PABC made the proposal because of its experience in this area (marketing the Clinical Support System MEDICL (AAP)) and also because there has been much published work on expert systems in the field of medical diagnosis. A prototype version of PARAMEDICL in a mixture of PARLOG and the sequential logic language PROLOG was implemented on the ICL PERQ (it was completed in December 1985).

1 Background

The field of medicine was chosen for the exemplar for this Fifth Generation prototype because there exists a considerable amount of published matter in this field, and there is a wide acceptance of the advantages of medical expert systems, of which a number have already been built. Indeed the International Health Unit, which is a part of Public Services Business, is marketing a clinical decision support system MEDICL (AAP).

The system described in this report was built in order to quantify the parallelism obtainable in an application. It was designed to be as flexible as possible and have in-built mechanisms for knowledge acquisition, and therefore methods of adding, modifying or deleting rules in a hierarchical structure of knowledge-bases. However, there was not sufficient time to design or implement a mechanism for modification, addition or deletion of rules based upon system analysis of the existing rules together with a

database of symptoms and final diagnoses of existing patients. Such an heuristic system would be a more ambitious and perhaps a future area of basic research.

2 MEDICL and PARAMEDICL

PARAMEDICL is an expert system written in a parallel logic language and intended to address the same application area as MEDICL (AAP).

MEDICL (AAP) – MEdical DIagnosis and Computer Learning (Acute Abdominal Pain) – is a computer based clinical decision support system designed to help hospital doctors in the diagnosis and management of patients with acute abdominal pain. It is not intended to replace the doctor in any way. It provides decision support teaching and feedback facilities.

Benefits of use include improved initial diagnostic accuracy leading to reduction in negative laparotomies (i.e. reduction in unnecessary surgery), reduction in perforated appendices and reduction in admissions and bed-nights.

MEDICL (AAP) is the most recent implementation of the project carried out by the Clinical Information Science Group at St. James Hospital, Leeds, under Mr. F.T. de Dombal, and results from a collaboration between ICL and the University of Leeds Industrial Services Ltd (ULIS). The database of 6000 previous cases and the teaching programmes used in this system were created by the World Organisation of Gastro Enterology (the OMGE) which has granted ICL a licence to use the database and has encouraged ICL in the development of MEDICL (AAP).

PARAMEDICL is so called to indicate that it is written in a parallel logic language and took as its inspiration MEDICL (AAP). However the prototype implementation is not based on the Bayesian matrix of probabilities used by MEDICL (AAP), nor is the internal design of the two systems related. The numbering system of symptoms as specified by the OMGE (Organisation Mondiale Gastro-Enterologie) and described in "Diagnosis of Acute Abdomen Pain" by Mr. F.T. de Dombal was used to facilitate a comparison of the two systems. (In the event, this comparison of systems never took place.) Apart from this common base of knowledge the two systems are very dissimilar. The prototype version of PARAMEDICL was implemented on the ICL PERQ, in which the parallel operations are simulated. The MEDICL (AAP) product runs on a DRS and is available as a supported product from ICL.

MEDICL is written in COBOL and uses Bayesian methods for calculating probabilities in order to arrive at its tentative diagnosis. Its database is a matrix of probabilities, constructed after careful analysis of a large number of case histories by the World Organisation of Gastro Enterology (OMGE), that gives the probabilities of a number of different symptoms being

associated with a number of different diseases – for example, of a patient with appendicitis experiencing pain in various sites. Given the symptoms observed in a patient and information and medical history obtained by interviewing the patient, the computer calculates the probabilities of the different diseases. The doctor can then use the computer system to confirm his own diagnosis, to look at crucial differences between his diagnosis and the computer's (if they differ), and at rare conditions which could possibly cause the patient's symptoms.

PARAMEDICL in contrast is rule-based and is written in the parallel logic language PARLOG. The set of rules that it uses are derived from published medical knowledge; it is not static, and rules can be added, deleted or changed.

For example, a pain-movement rule for appendicitis might be

```
if pain_present_site is RightLowerQuadrant and
pain_initial_site is CENTRAL then score 8
```

where "scores" are accumulated to give a measure of the likelihood that the patient is suffering from the illness in question, here appendicitis. In PARAMEDICL these scores were assigned intuitively by the authors and not based upon expert medical knowledge. In a real system the mechanism for favouring a medical diagnosis would have to derive from medical statistics.

PARAMEDICL's diagnosis proceeds by a question-and-answer process that can be conducted along any of three lines which we have called Normal, Specific and Opportunistic. These can best be explained by considering a hypothetical situation of a houseman and a number of specialist consultants ALL clustered around a patient's bed while the symptoms of the patient are elicited. This is modelled by representing each consultant as a knowledge-base-manager wherein reside a variable number of expert systems, in which each of a set of rules is represented in Horn-clause logic. The parallelism of the model is achieved by the "simultaneous" processing of symptom-messages by all knowledge-base-managers in parallel. Moreover all the expert systems in each knowledge-base-manager may process the symptom-messages in parallel with each other. The "real-world" situation is that all consultants consider all ramifications of every symptom in parallel with each other.

Only for a true parallel machine, or in the hypothetical situation of a set of specialist consultants clustered around a patient's bed, does the word "simultaneous" apply to the discussion here of "in parallel".

The three modes of operation of the model are described below, both in terms of the computer model and the hypothetical cluster of specialist consultants around the patient's bed.

The Model

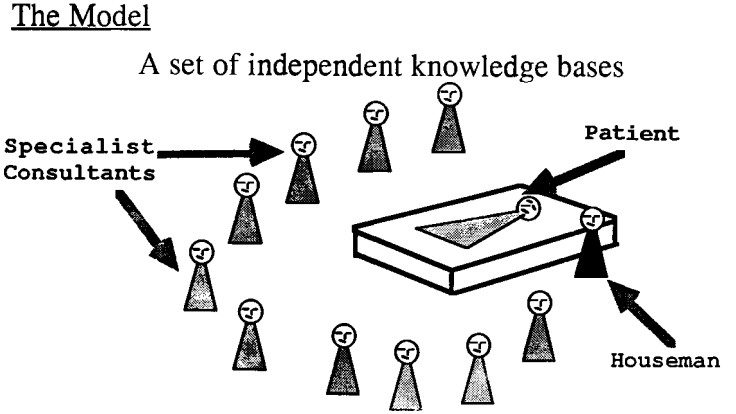


Fig. 1 Illustration of the model

Normal: This mode represents elicitation of symptoms sequentially by the houseman while the specialist consultants take a passive role listening for the indicators favourable or not favourable to the diagnoses for which they are the acknowledged experts.

The system presents an ordered sequence of questions to the clinician, and gives with each a set of possible answers. As each response is given to the computer it may initiate the application of rules for one or more expert systems in one or more knowledge-base-managers and result in the recalculation of a "score" for each disease under consideration. These "scores" are accumulated as the questioning proceeds and at the end of the session the doctor uses the final scores in whatever way he or she thinks appropriate. A very important feature – of MEDICL also – is that an explanation of any response can be asked for, so the doctor can always check or query the line of reasoning.

Specific: This mode represents following the choice of questions determined by a chosen knowledge-base-manager (consultant) considering only those factors, or symptoms, which are indicative of the disease(s) over which the specialist consultant is the acknowledged expert.

The computer offers to the doctor a choice of diagnoses so that the line of questioning may proceed with the aim of supporting the diagnosis that the doctor thought worth investigating (when the requested symptom is confirmed by the patient) or casting doubt upon the diagnosis (when the requested symptom is absent in the patient).

Opportunistic: In this mode the line of questioning follows that required to most effectively support the currently favoured diagnosis.

The computer questioning follows the sequence as for Normal but as soon as the scoring favours one diagnosis more than all others – which can be very

early on in the process – the system will nominate the knowledge-base-manager for that diagnosis to decide the subsequent questions. Subsequent responses may result in favouring a different diagnosis, in which case the selected knowledge-base-manager and therefore the line of questioning will change.

3 User view of PARAMEDICL

The PERQ prototype implementation was achieved in under three months. The system runs on a 2 Mbyte ICL PERQ with an A3 landscape screen. The current rule base contains about 1700 clauses coded in about 9300 lines of either PROLOG or PARLOG which are interpreted by the system – the PERQ does not have a PARLOG Compiler. The speed of execution on PERQ – on which all parallel operations are performed serially – is about 1600 logical inferences per second (1·6 Klips). If PARAMEDICL were run on a machine with a PARLOG compiler, the speed of execution would be an order of magnitude faster, and the mainstore requirement for PARAMEDICL would be significantly reduced.

For the question-and-answer process the screen is organised into a maximum of 13 windows showing

- the current question, with the possible answers
- the patient's history, accumulated as questioning proceeds
- a display for each of up to 11 knowledge-base-managers.

The mode of operation of the system is by selection from a menu. Thus initially a Patient History window is displayed in the top left corner of the PERQ screen which is empty. Each menu from which a selection may be made is displayed in the upper centre of the screen and a selection made therefrom using the puck. As each selection is made a representation of that symptom is added to the Patient History window which maintains the case notes of the patient. An example of a menu from which a selection is made is the "Pain-initial-site" menu from which a selection may be made of "Central" or "Right-Lower-Quadrant" or other locations as displayed in the menu window. This menu mechanism was chosen as an efficient mechanism for choosing only one selection from a set. Some symptoms can be found in combinations and a special multi-selection menu system had to be created for these.

Each consultant is represented as a knowledge-base-manager (kbm) which has its own window and position on the screen and into which information relevant to the diagnosis is displayed.

There are a number of system options. These allow the interruption of the normal flow of question and answer to facilitate the functions of adding, deleting, or modifying rules in any part of the system, or of obtaining explanation text from any expert system. There is also the option to change

mode from Normal to Opportunistic or Specific. (These modes of operation were explained in section 2.)

During the consultation with PARAMEDICL each kbm continually displays its status. Thus it is always possible to see the relative likelihood of each disease. Medical preference may dictate the hiding of this information until the user (who should be a responsible clinician) has independently come to a diagnosis. PARAMEDICL is a prototype which has not been subject to validation or to a clinical trial and was implemented as an open information system. No development of the prototype has taken place since the end of 1985 when the subject of this paper was completed.

4 Language Issues

The obvious choice for building an expert or diagnosis system was Prolog. With it we considered we could build a working prototype in the time available. However, Prolog is a language based upon sequential evaluation and is therefore not wholly suitable for parallel implementation.

PARLOG is also one of the target languages for the Flagship machine. There were three good reasons for choosing PARLOG as the language of implementation:

1 It is a "pure" declarative language very suitable for implementation on parallel architectures.
2 Our use of PARLOG would be among the first to exploit an extremely new language and any results which we had would be useful.
3 The implementation of PARLOG was on top of Prolog so we could always drop down into Prolog in the event of difficulty.

Our choice was therefore to use PARLOG as the language of implementation to establish the intercommunicating structure of the set of modules which comprised PARAMEDICL. A modular structure was considered crucial to the implementation of the protype within the proposed timeframe. Considerable advantage was obtained by designing for and using replicable shells for expert systems in this modular structure. Thus the system became dynamically modifiable because:

1 No assumptions were necessary about the order of execution of rules (whether interdependent or not).
2 Any rule could be dynamically altered in any way (including deletion) without affecting any other existing rules.

A measure of potential parallelism of the system could be calculated from the defined concurrency.

A single message representing a patient symptom and received by a knowledge-base-manager typically triggered around 20 concurrent pro-

cesses. Each kbm acted concurrently with all the other kbms; thus a parallelism of around 200 was not unusual. In addition, sets of messages could themselves be processed concurrently and these sets were typically of size 20. Despite the possibility of a parallelism of around 4000 we feel that PARAMEDICL is a small system and that future systems could exhibit far greater parallelism.

5 Knowledge acquisition

An aspect of PARAMEDICL not noticed as it was developed was its emergence as an effective knowledge acquisition tool. This was mainly due to the development of tools to facilitate faster development, the module structure, and common empty expert shells.

In a day-long session with a medical consultant from London we were able to evaluate the Human Computer Interface and both add new rules and modify existing rules in the system. The framework to support this activity allowed extremely rapid progress to be made to tune the system. The tuning of the PARAMEDICL prototype is facilitated by the ability to modify rules. This feature has considerable value in building up a system from nothing but would have to be used in a very controlled environment if a released system was able to be modified while in the middle of a diagnosis session.

The interactive nature of the system encourages its use for knowledge acquisition. It is extremely easy to modify or add rules as soon as the need for such change becomes evident. When developing rules a typical session would enter the details of patients whose eventual diagnosis was known, and then modify the rules to produce the expected profile of results.

6 System view of PARAMEDICL

The representation and organisation of the knowledge was very much the result of the iterative nature of the development method. Compared to a traditional Prolog system much more effort was put into how to structure the knowledge that was entered into the system.

This database was hierarchical. The top level system consisted of a number of smaller knowledge-bases, each of which was controlled by a knowledge-base-manager process. Each of these kbms knew about and controlled a number of small expert systems. The hierarchical nature of the system allowed for a far greater scalability, and any enhancement to the system could exploit this to encapsulate more detailed knowledge.

The system provided an interaction mechanism with which the user could interact with the different objects in the system. As the knowledge-bases and expert-systems were all identical in their classes, so were the interaction mechanisms enabling a simple but very effective interface to be developed.

Many similar knowledge bases and small expert systems

- all with default behaviour

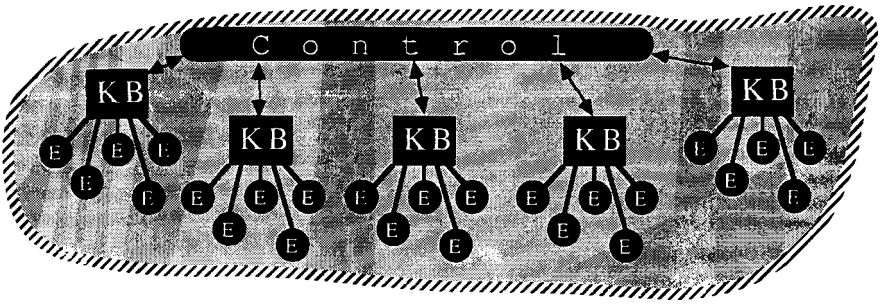


Fig. 2 Stereotypical knowledge representation. Many similar knowledge bases and small expert systems – all with default behaviour

The control of the various knowledge-bases was realised by treating them as separate processes with which the interaction process communicated. The process model developed was fairly limited due to the overhead of the interpreted PARLOG by Prolog. The design, however, allowed for the entire system to be represented as a network of communicating processes.

7 Future developments

We feel that the model provided by PARAMEDICL is extremely interesting because of the way it is possible to interactively and iteratively increase the amount and accuracy of the knowledge represented in the system. The iterative refinement of this knowledge might be a very time-consuming process. However, at present we do not have any data as to how "accurate" systems developed within this model could become.

The "output" of PARAMEDICL is a tailored knowledge-base about an **individual** patient with respect to a number of potential diagnoses. The measure of the system is the suitability and effectiveness of this knowledge-base to provide assistance in the diagnostic process.

The model on which PARAMEDICL is based could clearly be used for other diagnosis systems. The prototyping of any new system could be extremely rapid since the framework would be already in place – EPARAMEDICL (Empty PARAMEDICL).

A future development could be a many-level hierarchy of knowledge-bases and expert-systems. This would facilitate the structuring of knowledge-bases.

For example classifying a superset of diseases presenting themselves as appendicitis.

More efficient implementations of parallel logic languages are now becoming available. These provide potential for new advances in the development of highly interactive knowledge-based systems.

Acknowledgements

We should like to thank David Parry and Norman Brown for their encouragement and support, Mr. F.T. de Dombal (the originator of the MEDICL system specification and the author of "Diagnosis of Acute Abdominal Pain" on which much of the framework of PARAMEDICL is based) for several useful consultations, Dr. Chris Kibbler for a valuable knowledge acquisition session, and from Imperial College London, the PARLOG group and in particular Ian Foster for technical support.

Reference

1 DE DOMBAL, F.T.: 'Diagnosis of acute abdominal pain'. Churchill Livingstone, Edinburgh, 1980.
2 BURSTALL, R.M., McQUEEN, D.B., and SANNELLE, D.T.: *HOPE: An Experimental Applicative Language*, Department of Computer Science, University of Edinburgh, 1980.
3 TURNER, D.A.: *The Semantic Elegance of Applicative Languages*, pp. 85–92, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (Portsmouth, NH), 1981
4 CLOCKSIN, W.F. and MELLISH, C.S.: *Programming in Prolog*, Springer-Verlag, New York, 1981.
5 CLARK, K.L. and GREGORY, S.: "PARLOG: Parallel Programming in Logic," *Research Report DOC 85/5*, Department of Computing, Imperial College, London, May 1983.
6 GREGORY, S.: "Design, Application and Implementation of a Parallel Logic Programming Language," *Phd Thesis*, Department of Computing, Imperial College, London, 1985.
7 HOARE, C.A.R.: "Communicating Sequential Processes," *Communications of the ACM 21, 8*, pp. 666–677, 1978.
8 KAHN, K., MILLER, M.S. and BOBROW, D.G.: *Objects in Concurrent Logic Programming Languages*, ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, September 1986
9 SHAPIRO, E.: *Object Oriented Programming in Concurrect Prolog*, pp. 25–48, New Generation Computing 1, 1983.
10 UEDA, K.: "Guarded Horn Clauses," *Technical Report TR–103*, ICOT, Tokyo, 1985.
11 FURUKAWA, K., TAKEUCHI, A. and KUNIFUJI, S.: "Mandala: Knowledge Programming and System in the Logic-type Language," *Institute of New Generation Computer Technology*, vol. ICOT TR–043, February 1984.

S39XC – A configurer for Series 39 mainframe systems

C.W. Bartlett
ICL Knowledge Engineering Business Centre, Manchester

Abstract

In July, 1986, ICL Mainframe Systems released a rule-based system (S39XC) for the configuring of Series 39 Mainframes. This paper is concerned with the history of S39XC's development, its functionality, the methodology involved (both for development and delivery), its internal structure, and the general applicability of the techniques involved to other configuring problems.

Introduction

During recent years, starting with R1/XCON[1,2] and followed by OCEAN[3], ISC[4], MAPLE[5] and SYSCON[6], there has been a growing realisation that the problems of computer system configuration represent a fertile and profitable territory for the application of Knowledge Based System (KBS) techniques. Computer system configuration problems are instances of selection problems where the task is that of describing a single set of parts and connections, out of typically billions of "legal" combinations, and where this description meets user requirements which have been presented in comparatively high level terms.

The advantages to the supplier of computer systems of automating the process of configuration are several. For example:

There is a direct saving of effort, ratios of 16:1 being typical.

Bottlenecks due to shortages of human skilled effort can be reduced.

The quality of the computed configuration is many times better than that produced by hand.

The answering of "what if" questions becomes feasible, hence proposals can be much more closely adjusted to the customer's needs.

Losses due to incomplete configuring can be avoided (if the equipment supplied to a user is inadequate then the supplier bears the cost of making it good).

The possibility of lost orders due to over-quoting can be minimised.

In competitive situations, the supplier who can rapidly and accurately respond within hours to changing views of the user requirement has an obvious advantage.

The make-up of computer systems is subject to a high rate of innovation and product change. Problems arise in the accurate and timely communication of these changes to the sales force. There is also the problem of reworking the make-up of those systems which have been registered but not yet delivered.

Forward procurement and manufacturing can be based on high quality detailed information.

Cumulatively, these advantages can represent annual savings of several million pounds for the computer system supplier. Whilst the need for automatic configuring has always been known, attempts to meet this in the past by using conventional programming techniques have been largely unsuccessful, the main difficulty being that the time required to implement such systems is large in comparison with the rate of change and the duration of the product life cycle. It was only with the arrival of KBS, with its immense gains in productivity and reduction in timescales, that mechanisation of the configuring process became practical.

It was against this technical and financial background that the decision to implement such a computer system configurer, S39XC, was taken. The domain of S39XC is that of ICL Series 39 Mainframes[7] with their associated networks and bulk storage devices.

S39XC is intended for direct use by salespeople; it produces minimal, costed, detailed parts lists suitable for immediate input to ICL's Proposals and Contracts System.

History

In July 1984 a start was made on a prototype of a configurer for Series 39 Systems. This was initially written in ICL SAGE[8] and was worked on by several people. During this period no rigorous attempts were made to ensure that the knowledge contained was accurate, complete or up-to-date, the main objective being to demonstrate the potential and viability of the techniques. By February 1985 work had progressed to the point where an adequate demonstration of the possibilities of KBS was available to ICL management. This led to a decision to proceed with a full scale working version.

Initially work was confined to translating the original prototype to the newly available ADVISER[14] language and to some experimentation with the form and style of the Human Machine Interface (HMI). This early work was

conducted on an informal basis and led to an improved understanding of the nature of the problem. In October of 1985 ICL's Knowledge Engineering Business Centre entered into a formal commitment with Mainframe Systems for the production of a full-scale working version suitable for direct use by the sales force. Work on the final version, S39XC, commenced early in November of 1985.

An initial re-examination of the structure of the problem led to a decision to re-implement the whole system rather than take the prototype as a starting point. There were three main considerations behind this: first, the packaging of the basic core components of Series 39 Systems had been radically changed (with the beneficial effect of rendering the problem monotonic); the structure of the prototype was not suitable for the incorporation of various features which had been identified as required in the final version; and the prototype had been converted from SAGE to ADVISER – as a consequence full use was not being made of the facilities of the ADVISER language.

With these issues settled, writing of the actual code for the system started in mid-November 1985. By mid-April 1986 S39XC was made available to a restricted set of (relatively informed) users on a field trial basis. During this period the only criticism received was in the area of the HMI and the facilities available. No cases of errors in the configuring process were found during this field trial. By this time S39XC contained some 2750 rule equivalents (this term is used since some of the ADVISER single constructs are equivalent to several standard "If ... Then ..." rules) expressed in about 9500 lines of source. This had been written in 90 working days and had taken about 150 mandays of combined Knowledge Engineer (KE) and Domain Specialist (DS) effort.

Early in July 1986, with the final points arising from the field trial attended to, S39XC went live for general use by the ICL sales force. By late-November 1986 the system was being used on a regular basis by some 400 people and some 1000 configurations had been registered. It is estimated that these 1000 configurations represent about 4000 uses of the system. None of the users of the system were given any documentation or training in the use of the system other than minimal instructions on how to access the service and a guide to what it did. So far there have been no confirmed instances of misconfiguration.

With the first release Mainframe Systems assumed formal Design Control Authority for S39XC; much of the subsequent updating of the system has been by the DS himself, with the KE and the Knowledge Engineering Business Centre taking a subordinate consultancy role.

Since the first release there have been a number of further issues extending the coverage of the system. S39XC now has about 4600 rule equivalents contained in some 15 500 lines of source.

Functionality

Role

S39XC acts in a number of roles. During the initial stages of order taking S39XC performs as an adviser, allowing the salesperson to describe alternative systems and have them evaluated as to content, cost and technical suitability. The results of these investigations are immediately available on paper.

Once a firm requirement has been reached S39XC acts as a technical sales clerk, taking the high level description of the requirement, expanding it to an accurate, complete low level description and automatically entering it into the Proposals and Contracts System. Quality is assured during this stage by preventing the salesperson from, in any way, modifying the expanded description.

Subsequent to an order being placed S39XC may act again in the role of a technical sales clerk if, for example, changes occur in the details of the various parts and the relationships between them. In this case S39XC is run in batch mode to reprocess the requirement without any further interaction with the salesperson.

Further, experience gained with running S39XC shows that it also to some extent acts in the role of a trainer. This in fact was not a preconsidered role but probably comes about from the ease of use and the copious explanation and help available.

Scope

S39XC is concerned with configuring ICL Series 39 Mainframe Systems. In respect of this, the following areas are covered:

Node requirements: Concerned with the selection of the basic type and numbers of OCPs, the amount of storage, the location of the system's CAFS-ISP[9] and the requirements for basic (bundled) software.

FDS300 requirements: Covering the requirements for FDS300 Controllers and Drives, dual connections and the numbers of extra CAFS-ISP required. Additionally, the implications of adding MDSS type disc systems are investigated in this stage.

FDS2500 requirements: Similar to FDS300, covering requirements for Controllers, Drives, dualling and CAFS-ISP.

Retained discs: Disc systems which derive from ICL 2900 and ME29 ranges are potentially transferable to Series 39 Systems. This area is concerned with selecting the appropriate parts relevant to such transfers.

Series 39 Magnetic Tapes: Catering for the attachment of Series 39 Magnetic Tapes.

Retained Magnetic Tapes: As with Discs, it is possible to transfer existing Magnetic Tapes to Series 39. Suitable interfacing parts need to be specified.

MACROLAN: The MACROLAN[10,11] is the method by which the processing nodes and the bulk storage devices are interconnected as a network. This area is concerned with identifying the parts and cables required to make these interconnections.

Specific communications requirements: This area is concerned with identifying those parts related to supporting various mandatory (e.g. Teleservice) and specific (e.g. X25) services.

General communications requirements: This is primarily concerned with those parts required to support communication with other systems.

Printer requirements: A wide range of Printers is available for Series 39. Associated with each type of printer is a range of options concerned with the character set size and make-up, including Language Variants.

Slow-speed retained devices: Certain slow equipment (e.g. printers, card readers) may be retained from previous ICL ranges. Special interfacing parts are required to allow their attachment to Series 39.

OSLAN: The OSLAN[11] is the network which interconnects the various communications outlets and slow devices. OSLAN networks can be of considerable size and complexity. At the time of writing S39XC configures only the small and simple requirements, the salesperson being advised to seek expert human assistance for the more difficult cases. However, even in these cases S39XC will provide the minimum subset of the parts required.

Mandatory software: The set of requirements described by the user needs a minimum set of microcode and Operating System software to complement it. This is evaluated by S39XC.

Optional software: Over and above the software required to make the system work, the salesperson is able to specify requirements for various enhancements and options.

Main functions

S39XC is a selection system. It selects that system configuration, out of some 1 000 000 000 000 000 000 possibilities, that most closely meets the requirements. In contrast to some systems[1,2], S39XC does not perform any physical layout functions. Problems of physical layout in Series 39 systems were attended to at the design stage, and anyway are not particularly troublesome in a distributed system.

S39XC also provides "expert criticism" of the user's intentions. For example if there is a mismatch between the amount of bulk storage and the processing capability of the system then the user would be advised:

> "You have specified more Disc and Magnetic Tape controllers than would be expected for a node with this level of power. You need to do an overall sizing of your I/O requirements to ensure that the system will cope with the projected workload."

These annotations become attached to the order and the order may be blocked until they are cleared or specific Marketing approval obtained.

Additional features

During the process of determining the configuration S39XC is able to deduce that the user is intending to supply other equipment which is outside the scope of S39XC. The user is reminded of the need to attend to these extra requirements and this reminder is also attached to his order. Orders which have these reminders attached require specific confirmation that the extra requirements are being attended to.

An important feature of S39XC is its Help System. Following a proposal arising out of the Eurohelp project[12,13], this is structured into three components:

The "Librarian";
The "Butler";
The "Guardian".

The Librarian is an information provider covering such topics as:

How to answer questions;
The boundaries of the system;
Known restrictions;
Content of standard Series 39 packages;
Teleservice requirements;
Bug reporting;
Contacts for expert human assistance.

Additionally, the Librarian is able to offer advice and explanation in respect of all questions asked of the user. To a limited extent, i.e. restrained to the context of the conversation up to the time of invocation, some of these explanations may have been tailored to match the individual consultation. Note that whilst the underlying systems used to construct S39XC provide the conventional How and Why facilities these are regarded as being of primary interest only to the DS and KE. Thus they are not provided as part of the Librarian service and are in fact masked off from access by the normal user.

The system, ICL ADVISER[14], used to implement S39XC, provides a comprehensive range of low level commands for the control of the consultation. Whilst invaluable to the DS and KE, their use by the salesperson would be difficult since it would involve a high degree of understanding of the way in which ADVISER and S39XC work. Hence access to these functions is masked off and instead *the Butler* is available to perform the following actions on the user's behalf:

Begin the consultation all over again;
Restart the consultation at a previous point;
Quit the consultation;
Save the current state of consultation.

The "Save" facility is particularly important. It allows the user to store the point in the consultation reached when there is, for example, a telephone interruption. On re-entry to S39XC the Help system will notice the presence of a stored position and offer the user the option of resuming from the point of interruption.

At the end of a complete run, the Butler automatically offers the user the possibility of saving the (high level) details of the consultation. On subsequent entry to S39XC, if the presence of any such saved details is noted then the Butler will offer the user the option of selecting one of these files for restoration. In this case, the user is then offered the option of having his responses processed immediately to the low level expansion or of stepping through them question by question. In this latter case, whenever a question is asked the previous reply is retrieved and the user may, if so desired, change it before sending it back. Note that if changes are made then some of the previous session's replies may be rendered invalid by the propagation of any relevant constraints. This will lead to S39XC automatically requiring the user to also change these further invalid replies – thereby maintaining the integrity of the generated parts list.

A further facility offered by the Butler is an option for the user to examine the "cost tree" associated with the total configuration price. This option is automatically offered if the user has asked for a costed configuration.

The Guardians are a set of pro-active agents which continually monitor the internal state of the consultation for constraint violation. When such a violation is detected the consultation is immediately interrupted and an informatory message is displayed indicating the root cause of the problem. On resumption, the user is taken back in the question sequence to a point relevant to recovery from the problem. There are some 25 of these pro-active agents in S39XC, covering a range of problems from simple cases like attempts to dual an odd number of data channels to the more exotic constraint violations which may occur during, for example, configuration of the OSLAN.

Also conceptually part of the Guardian system, but not explicitly identified as such to the user, are a set of provisions whereby the acceptable boundaries on the reply to any question seeking a numeric response are adjusted to match the constraints of the configuring process and the state of the consultation to date. To avoid "cueing" the user, these boundaries are not initially visible and are only made visible should a violation be detected. If the reply to a question lies outside the relevant boundaries then the question is immediately re-asked, this time with the boundary values being made visible to the user.

Delivery

S39XC is available to its users as a MAC Service on a single central service. Users connect to the service either by direct dial-in or by using the facilities of ICL's Network. The majority of the users use a One Per Desk as their terminal – much of the rapid take-up of S39XC can be attributed to the widespread availability of OPDs amongst ICL's salespeople.

Methodology

Knowledge elicitation

Contrary to the industry folklore[15,16,17], elicitation of the knowledge required for successful configuring was not particularly difficult or time consuming. That it was successful is evidenced by the high quality obtained in the final product.

However, there were a number of special factors mitigating the elicitation task:

The KE was already reasonably informed as to the general details of Series 39 make-up and architecture;

It was possible to structure the problem into agreeably sized areas which could be tackled essentially independently of each other, with little recapitulation being required;

The problem was clearly defined and did not involve any uncertain reasoning.

During the course of the development of S39XC the DS was given a condensed version of the ADVISER Language Course. Whilst primarily intended to prepare the DS for the time when S39XC was handed over to Mainframes, this training had the noticeable side effect of changing and improving the nature of the dialogue between the DS and KE. As a follow on to this training and in order to reinforce his understanding of the ADVISER Language, the DS undertook the writing of the initial version of one of the areas himself without any intermediate elicitation. The DS also directly inserted all of the explanatory text associated with each question.

An important function performed by the DS was to act as a single focus for knowledge being passed to the KE, thereby freeing the KE from any need to resolve disputes between conflicting DSs. It did however put some of the burden of knowledge elicitation on the DS himself.

Rather than attempt the mammoth task of eliciting the whole of the knowledge in one session, the strategy adopted was to work through the various hardware areas one by one. The typical pattern for any one area was for the DS and KE to interact over a period of one to two days concerning the requirements for the area. Following this, over a period of 5 to 10 days the KE would encode these requirements and perform a thorough test to ensure that this formal description accurately represented his (i.e. the KE's) understanding of the requirement. Once confidence was achieved at this level, the area was formally passed to the DS for further testing, this time to ensure that the area performed as required to solve the problem. This cycle was repeated until both the KE and the DS were satisfied. In practice, the cycles for individual areas were overlapped, thus e.g. whilst the DS was testing the FDS300 area the KE was working on the FDS2500 area.

Although rarely mentioned in the literature, the importance of this extensive testing by the DS cannot be over-emphasised.

Construction

From the start, it was realised that the main issue relevant to the construction of S39XC was that of quality. It was realised that with KBS techniques being totally new to the main body of users any initial failures could easily prejudice them against the whole concept of KBS. Issues concerned with coverage and implementation timescales, whilst important, were therefore treated as subordinate to quality. In the event, implementation timescales were not a problem; and with one of the consequences of the emphasis on quality being a careful attention to the internal structure of S39XC, expansion of the coverage has not caused any problems.

The methods used to achieve quality were no different from the industry standard techniques of software engineering: careful partitioning and subdivision of the problem; a rigorous, planned testing schedule; control of the interfaces between partitions; detailed written (double entry) book-keeping; and controlled debugging.

Although the initial prototyping of S39XC had been conducted in ICL SAGE, before fully committing to use of ICL ADVISER some thought was given to development methods based on LISP or PROLOG, with (possibly) later conversion to a more suitable delivery route. Other issues considered at this time were concerned with whether ICL ADVISER was adequate as it stood or would its functionality need extending by significant amounts of conventional program.

Quality was the main issue in deciding this question. Whilst it would have been possible to deliver applications written in ICL LISP or ICL PROLOG given the size of the application it was fairly obvious that the performance would not match the user requirement and hence some sort of conversion exercise would be required. This however would have led to serious problems with respect to establishing and proving the quality of the conversion route. In contrast to this, applications developed using ICL ADVISER require no conversion before delivery. Further, there was good evidence from the prototype that the functionality of ICL ADVISER was adequate. In the event, only some 40 or so lines of conventional code were required – in an area concerned with handling external files.

Development of ICL ADVISER applications is a two stage process. Source is first processed by the Builder into a computationally efficient representation of the inference net. During this Building phase, extensive syntax and certain semantic checks are applied. Having passed this phase, the Interpreter is used to animate the knowledge and present it to the user.

Knowledge is represented in ICL ADVISER by Facts and Rules. Facts are typed, with Assertions, Integers, Objects, Strings, Records and Arrays being available. Rules in ICL ADVISER are inferences, with Facts becoming instantiated once and once only. For control purposes there is a facility to return a Fact to the unevaluated state, but the value of a Fact cannot be directly changed without this intermediate stage. Inference in ICL ADVISER is primarily by means of backward chaining, with forward chaining being used to control the dialogue sequence and the cohesiveness of interim and final reports.

ICL ADVISER supports the structuring of problems by providing an Area construct. An Area is similar in concept to the Module of conventional programming and is provided with formal Interface Definition constructs.

The achievement of high quality was very much eased by the declarative, single assignment style of programming supported by ICL ADVISER. Although there were some control issues to be faced in the implementation – and these are well known danger areas – they were very few and, since they were not confused with the actual process of configuring, could be concentrated on in depth.

As illustrated in Fig. 1, S39XC is divided into a number of distinct ADVISER Areas. This division is primarily organised to enable the required control facilities and options to be easily implemented. Secondary to this is a desire to achieve a clean functional distinction between the various parts of S39XC. Note, though, that these two aims are not necessarily contradictory.

The Configurer Area (1) is concerned with:

(a) Collecting general user and run details;

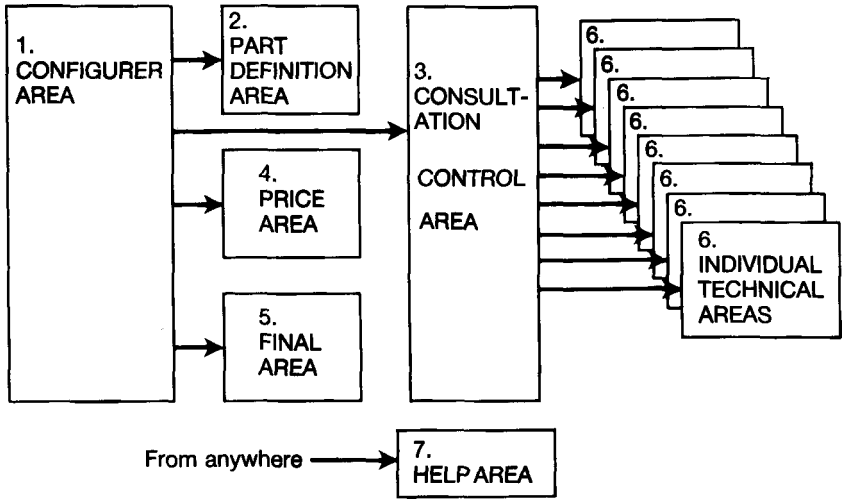


Fig. 1 The structure of S39XC

(b) Offering restart from interrupted sessions or from old cases;
(c) Setting up the required type of run;
(d) Activating (indirectly) the technical areas;
(e) Offering the opportunity to Save the last case;
(f) Offering various end-of-run options.

The Parts Definition Area (2) acts as a depository for facts concerning the parts available for selection and their prices. It is also able to acquire these prices from an external file if so required.

The Consultation Control Area (3) is concerned with the sequencing of entry of individual technical areas, interspersing these entries with advisory/help text as required.

The Price Area (4) offers browsing facilities with respect to the details of how the total price of the configuration is derived. It is activated only if a priced configuration is being generated.

The Final Report Area (5) is where the results of the whole consultation are summarised and presented to the user. This area also contains the heuristics used to generate the "stylistic" criticism of the composite configuration. If required, the user is also given practical advice on such topics as how to take a screen copy of the output of the run.

The Individual Technical Areas (6) are where the actual configuration work is performed. S39XC contains eleven such areas each with quite varied sets of rules dependent on the nature of the specific technical problem. However, each Area in this set also conforms to a common thematic structure of conducting the specific conversation, evaluation of the relevant part counts,

production of an Area summary, and presentation of an option to re-run the Area again. Closely similar mechanisms are used to achieve these effects in each individual Technical Area.

The Help Area (7) may be entered at any time. It contains the Librarian and Butler facilities. The construction of the Help system was a little unusual, a fair proportion of it consisted of the KE encoding his own knowledge of the internal structure of S39XC – an interesting example of self-reference[18].

Future

The next stages in the development of S39XC are likely to be concerned with its extension into superstructure software and the generation of French and German language variants.

In the area of mainframes, the process of selecting configurations is likely to be enhanced by a precursor which is able to size the customer's requirements, possibly along the lines of DRAGON[19], and provide direct input to S39XC in respect of many of its required values.

Taking a broader view within ICL, it has been decided to include a configuring system as part of the basic development requirements for new products.

Future work within the Knowledge Engineering Business Centre will include an examination of the case for the development of tools systems[20] specifically targetted at the production of KBS configurers. Such a tools system would be aimed at configuring problems in general, e.g. conveyor systems, power drive systems, electrical distribution systems[21], rather than at the relatively narrow domain of computing.

Conclusions

S39XC has been a remarkable success. The reasons for this success are of relevance to most KBS projects. To summarise:

(a) A highly geared business case could be made out;
(b) It could be demonstrated that conventional techniques were not suitable;
(c) There was an existing community of needful users;
(d) The domain was eminently suitable in that it was clearly bounded, was of a suitable "grain" size and did not involve inexact reasoning;
(e) The technical means for delivery to a large community were already in place;
(f) Given that the problem was to be tackled by the use of KBS techniques then it was possible with only two people involved. Thus many of the difficulties caused by multiple communication channels were avoided;
(g) A suitable "shell" was available. Its use meant that all of the intellectual effort of the KE and DS was focussed on the main issue;

(h) Finally, the lessons of software engineering so painfully learned in conventional programming are just as applicable to KBS applications – though they may need some re-interpretation.

Acknowledgements

Many people contributed to the success of the S39XC project, not least amongst these being the ADVISER implementation team (Dave Mitcalf and Linda Hughes) whose painstaking devotion to quality so much eased the construction of S39XC. Thanks are also due to those GIS people whose enthusiasm paved the way for a rapid introduction of the service to a world-wide user base. That the project was ever started at all is due to the visionary encouragement and risk taking of the management of Mainframe Systems and of the Knowledge Engineering Business Centre. Credit is also due to Derek Sayers, Ken McLauchlan, Alan Pooley and Karen Morley whose early prototyping demonstrated the possibilities for the construction of S39XC.

It almost goes without saying that but for the remarkable enthusiasm and devotion of the Domain Expert, A. ("Sandy") W. Smith, S39XC could not have existed at all.

References

1 McDERMOTT, J.: 'R1: A Rule-based Configurer of Computer Systems'. Technical Report CMU-CS-80-119, Carnegie Mellon University, Pittsburgh, PA.
2 McDERMOTT, J.: 'R1: An Expert in the Computer Systems Domain'. Proc. First Nat. Conf. AAAI, 269–271.
3 SZOLOVITS, P. and CLANCEY, W.: 'Case Study: OCEAN'. Tutorial 8, IJCAI, 1985.
4 WU, H., VIRDHAGRISWARAN, S., CHUN, H.W. and MIMO, A.: 'ISC – An Expert System for the Configuration of DPS-6 Software Systems'. 9th Annual Honeywell International Computer Sciences Conference, 1985.
5 BOWEN, J.: 'Automated Configuration using a Functional Reasoning Approach'. Proc. AISB-85, U. Warwick, April 1985.
6 ROLSTON, D.: 'An Expert System for DPS 90 Configuration'. 9th Annual Honeywell International Computer Sciences Conference, 1985.
7 SKELTON, C.J.: 'Overview of the ICL Series 39 Level 30 System'. ICL Tech. J., Vol. 4, Iss. 3, May 1985, 225–235.
8 'ICL SAGE User Guide'. ICL Publication RPO480.
9 MACPHAIL, N.: 'Development of the CAFS-ISP Controller Product for the Series 29 and 39 Systems'. ICL Tech. J., Vol. 4, Iss. 4, May 1983, 393–401.
10 STEVENS, R.W.: 'Macrolan: A High-Performance Network'. ICL Tech. J., Vol. 3, Iss. 3, May 1983, 289–296.
11 BROUGHTON, P.: 'Input/Output Controller and Local Area Networks of the ICL Series 39 Level 30'. ICL Tech. J., Vol. 4, Iss. 3, May 1985, 260–269.
12 HARTLEY, R.J. (U. Leeds, Dept. Comp. Based Learning): 'The Rationale of Explainer'. Eurohelp Project, CBLU-ULE/EUROHELP/024, June 1986.
13 BREUKER, J. and DE GREEF, P. (U. Amsterdam, Dept. Social Science Informatics): Deliverabe 12.1, ESPRIT Project P280, 'EUROHELP', December 1985.
14 'ADVISER User Manual'. ICL Publication R30008/02, 1985.
15 HAYES-ROTH, F., WATERMAN, D.A. and LENAT, D.B.: 'Building Expert Systems'. Addison Wesley, 1983, 129.

16 DUDA, R.O. and SHORTCLIFFE, E.H.: 'Expert Systems Research'. Science, 1983, 220, 265.
17 WILKINS, D.C., BUCHANAN, B.G. and CLANCEY, W.J.: 'Inferring an Expert's Reasoning by Watching'. Proc. Conf. Intelligent Systems and Machines, 1984, 1.
18 HOFSTADTER, D.: 'Godel, Escher, Bach: an Eternal Golden Braid'. Penguin Books.
19 KEEN, M.J.R.: 'DRAGON: The Development of an Expert Sizing System'. ICL Tech. J., Vol. 3, Iss. 4, November 1983.
20 WU, H., CHUN, H.W. and MIMO, A.: 'ISCS – A Tool Kit for Constructing Knowledge-Based System Configurators'. Proc. Fifth Nat. Conf. Art. Int., Philadelphia, Aug. 1986, 1015–1021.
21 BORGHESI, M., GIORGESI, R. and DECIO, E.: 'An Application of Knowledge-Based Systems Technology to Configuration Problems'. Proc. First Int. Expert Systems Conf., London, Oct. 1985, 99–118.

Appendix A – example S39XC source

Rules

ICL ADVISER offers a powerful syntax for expressing rules, allowing the clustering together in a compact form the rules concerned with a particular concept. Thus each rule in the following example has the structure:

```
(If ... Then (If ... Then ...
                    Else (If ... Then ...
                                Else ...)
```

```
INTEGER extra_mpsu:
    "The number of extra mpsu required to form the network."
RULE set_extra_mpsu_1_node: "for the single node case"
  PROVIDED no_of_nodes = 1
    extra_mpsu IS 0
  PROVIDED 5 * mpsu_ct > controller_ct
ALSO
    extra_mpsu IS (controller_ct - 5 * mpsu_ct) DIV 4
    PROVIDED (controller_ct - 5 * mpsu_ct) MOD 4 = 0
ALSO
    extra_mpsu IS (controller_ct - 5 * mpsu_ct) DIV 4 + 1

RULE set_extra_mpsu_2_node : " "
PROVIDED no_of_nodes = 2
    extra_mpsu IS 0
PROVIDED 4 * mpsu_ct > controller_ct
ALSO
    extra_mpsu IS (controller_ct - 4 * mpsu_ct) DIV 4
PROVIDED (controller_ct - 4 * mpsu_ct) MOD 4 = 0
ALSO
    extra_mpsu IS (controller_ct - 4 * mpsu_ct) DIV 4 + 1
```

The above example captures the count of the number of extra MPSUs required to allow the construction of the network, over and above those

required for the connection of controllers. There are two main cases, dependent on the number of processing nodes present in the system.

Demons

The mechanism by which the pro-active elements of the Guardian are implemented is the ADVISER Demon. The following example illustrates this:

First we have an Assertion (a fact which can have gradations of values between True and False) which has an associated rule which expresses the illegal condition of interest. In this case, this is where the user has specified a number of drop cables which is incompatible with previous requests.

```
ASSERTION illegal_da_cable_ct :
    "the number of da cables is incorrect"
RULE set_illegal_da_cable_ct :
    "if the number of cables does not match the equipment count"
    illegal_no_da_cables IS TRUE
PROVIDED
    (SUM(da_cables_required)   <    (  transceiver_ct
                                     + devices_via_concentrator
                                     + repeater_cable_count))
OR (SUM(da_cables_required)  >2 * (  transceiver_ct
                                     + devices_via_concentrator
                                     + repeater_cable_count))
```

Watching this Assertion we have a Demon which is triggered if, and only if, the Assertion becomes instantiated to True.

```
DEMON watch_da_cable_ct : "for inconsistency"
WHEN illegal_da_cable_ct
ADVISE  " I N C O N S I S T E N C Y "
    , " You must specify at least " ,    (  transceiver_ct
                                          + repeater_cable_ct
                                          + devices_via_concentrator)
    , " and no more than " ,          2 * (  transceiver_ct
                                          + repeater_cable_ct
                                          + devices_via_concentrator)
    , " Drop Cables.  Please try again. "

ALSO WIPE  ask_da_cables_required
         , da_cables_required
         , illegal_da_cable_ct

ALSO CONSIDER da_cables_required
```

If the Demon is activated it informs the user the nature of the problem, returns the relevant facts to the Unevaluated state and then causes the user to be asked to resupply the requirement for Drop Cables.

Appendix B – example run

Although written as a MAC application, use of S39XC gives much of the appearance of a TP application – both in its speed of response and in its total use of a screen mode of presentation. Note though, the majority of the screens have a content which is a function of the foregoing progress of the interaction.

The following extract from a run of S39XC shows the dialogue and interim report concerned with the selection of FDS300s. First the user is asked to specify the number of banks of FDS300 required. If none the user may type 0 or merely press (SEND). In the latter case a default value will be supplied by S39XC.

```
SERIES 39 EXPERT CONFIGURER          Type !HELP for help
                                     Type !EXP  for clarification
 TASK 2 : Configuring the MACROLAN
-------------------------------------------------------------------
FDS300 REQUIREMENTS

NOTE: FDS300's attached to MDSS controllers are configured
      later as part of Retained Disc Configuring.
      DO NOT INCLUDE THESE REQUIREMENTS HERE

A bank of FDS300 is a group of interconnected drives
excluding the controller. It is therefore a set of drives
attached to an FDS300 Master Module including the two
drives housed within the Master Module.

How many banks of FDS300 do you require?   )4      (
Please type the number you require and press (SEND).

If you don't understand what is meant by "bank" then
                        type EXP then (SEND) for an explanation.
```

Next the user is asked to specify the number of drives for each Bank. Note that the number of lines in the menu matches the previous reply.

```
SERIES 39 EXPERT CONFIGURER          Type !HELP for help
                                     Type !EXP  for clarification
 TASK 2 : Configuring the MACROLAN
-------------------------------------------------------------------
FDS300 REQUIREMENTS

Please supply your requirements for FDS300 discs.
You can have a minimum of 2 discs and maximum of 8 on any bank.

Fill in the number of discs you want against each bank,
                  then press (HOME), then (SEND)


 Bank    1 : )3     (
 Bank    2 : )4     (
 Bank    3 : )3     (
 Bank    4 : )5     (
```

The user is next questioned regarding the requirements for dualling the Banks of FDS300. Note the user is given a reminder of the number of drives selected for each Bank.

```
SERIES 39 EXPERT CONFIGURER          Type !HELP for help
                                     Type !EXP  for clarification
 TASK 2 : Configuring the MACROLAN
-------------------------------------------------------------------
FDS300 REQUIREMENTS

 Please indicate which banks are to have dual access.
      NOTE : You must choose an even number of banks.

 Make your choices by typing Y against the banks
 you wish to have dual access and
 then press (HOME), then (SEND).


 Bank    1 ( 3 discs) : )Y     (
 Bank    2 ( 4 discs) : )      (
 Bank    3 ( 3 discs) : )      (
 Bank    4 ( 5 discs) : )Y     (
```

In the next screen, a simple yes/no response is used in respect to the requirement for MDSS. A reply of "Yes" would lead to a set of screens relevant to this requirement.

```
SERIES 39 EXPERT CONFIGURER          Type !HELP for help
                                     Type !EXP  for clarification
 TASK 2 : Configuring the MACROLAN
-------------------------------------------------------------------
FDS300 REQUIREMENTS



Do you want to add any MDSS drives to FDS300? )NO      (
Please type Y or N and press (SEND)
```

Here, the user is reminded that the basic package selected already contains one CAFS and is asked to supply the requirement (if any) for further units.

```
SERIES 39 EXPERT CONFIGURER          Type !HELP for help
                                     Type !EXP  for clarification
 TASK 2 : Configuring the MACROLAN
-------------------------------------------------------------------
FDS300 REQUIREMENTS

NOTE : You have already selected CAFS
       as part of your basic configuration.


How many additional CAFS units do you require? )2      (
Please type the number you require and press (SEND)
```

The area concludes with a summary of the parts selected to meet the FDS300 requirement. If the user had previously selected pricing then this screen would also contain details of the Sales Price and Quarterly Engineering Service Charge, together with their totals.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                  F D S 3 0 0   C O N F I G U R A T I O N           +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

4 x 003631/04 - FDS300 MASTER TYPE 2
2 x F03654/05 - FDS300 CAFS-ISP
2 x 003632/01 - FDS300 MODULE 1
1 x 003632/02 - FDS300 MODULE 2
1 x 003632/03 - FDS300 MODULE 3
2 x F03654/21 - FDS300 2ND COUPLER
1 x F03654/34 - FDS300 DA CBL DR 1-6(X2)
4 x F03654/31 - FDS300 SA CBL DR 3 - 6

Are you satisfied with the FDS300 specification ? )YES    (

(If you answer N you will be asked to respecify the FDS300)
```

At the end of each area, the user is given the opportunity to re-enter the area at the start and re-answer the questions. It is thus possible to cycle round until the area is acceptable. When a "Yes" reply is given to the final question the FDS300 parts are transferred to the final total selection and any constraints generated are carried forward to the next area.

The application of knowledge based systems to computer capacity management

M. Small
ICL Knowledge Engineering Business Unit, Manchester

Introduction

Capacity Planning is very much like budgetary planning in that its objective is to share out the resources of the computer system both by quantity and by timescale though possibly to a timetable calculated in hours as well as in weeks and months.

Just like any plan it is reliant upon the data supplied by those participating in the plan as much as or probably more than those supplied by outsiders. So for instance just as most budgetary plans rely upon reasonable estimates of the exchange rate and the bank rate, capacity plans rely upon the capabilities of processors and routes. However, just as incorrect sales revenue forecasts are likely to have far reaching implications on the budgetary plans, so too have incorrect demand forecasts on any capacity plans.

In building a capacity plan an expert practitioner goes through a systematic process. This process is one of collecting and assessing the information concerning the existing and proposed demands made on the computer system together with their impact on that system. This process is usually called sizing.

Sizing is not an exact mathematical science although it may involve the use of mathematical and statistical techniques. Successful sizing involves the use of practical experience and expert knowledge. Not everyone is equally skilled at sizing and indeed there are recognised experts in this field. However the knowledge used by these experts, whilst complex, is highly specialised. These factors taken together indicate that computer systems incorporating this specialist knowledge could be produced, using knowledge engineering techniques.

Three systems, relevant to this area, have been produced by ICL using knowledge engineering techniques and technology. These systems are:

DRAGON – an expert system for sizing new workloads.

S39XC – an expert system for configuring Series 39 systems. This is described in another paper in this journal[5].

VCMS – a Capacity Management System for VME.

What is knowledge engineering?

Knowledge engineering has its roots in that branch of computer science known as "artificial intelligence", which can be summarised as an attempt to develop computer systems which in some way mimic human intelligence. Experience gained in following these goals, and the advances in power of computer systems, have now led to the point where it is possible in many fields to create systems with a problem solving capacity as good as (and in some cases better than) that of a human expert in that field.

Such systems are known, appropriately enough, as "Expert Systems" (other names used are Knowledge Based Systems or even Knowledge Based Expert Systems). An early aim of researchers in artificial intelligence was the construction of general purpose problem solvers; in contrast Expert Systems represent the realisation that the best way to solve real-world problems with computers is to incorporate the knowledge and experience of human experts. An Expert System therefore solves problems (or provides guidance or training) in a particular area by making available to the computer the skills of an expert in that area.

KNOWLEDGE ENGINEERING, then, is concerned with

- providing means of storing or representing human knowledge – this is usually known as the "knowledge base"
- providing techniques for getting knowledge out of the expert and into the knowledge base in an appropriately structured form – this is called "knowledge acquisition"
- providing the means by which the user can state or examine his problem and arrive at the solution by reference to the knowledge base – in the jargon of the industry this is known as the "inference engine"
- doing all this in an easy-to-use, "user friendly" manner.

Why not use traditional methods?

It would be possible, given enough time, to construct an expert system program in almost any computer language from BASIC to COBOL, FORTRAN to PASCAL. But the problems would be enormous. First of all, there would be no real computer assistance in the knowledge acquisition – it would be a matter of pencil and paper. Then that knowledge would have to be turned into code in a program, a difficult task requiring programming expertise. If the knowledge was right and the code was accurate the program would work (it might not have a very easy user interface, but that is another matter), and it could be tested on real problems. If it was then realised that the knowledge was wrong, the whole structure of the program might have to

be changed – an extensive re-coding of the program. And when the program was complete and working properly it would be a program capable of solving only one problem area – to construct another expert system for another area, even a related one, would involve starting from scratch again.

Contrast that with the Knowledge Engineering approach. The vital difference here is that the knowledge is held as "data", not as "code". The person constructing the system starts off with an empty expert system "shell". Into this shell he can put expert knowledge piece by piece as he acquires it, and can test it and if necessary correct it as he goes along, until he is satisfied that he has all the knowledge he needs from his expert and that it has been correctly put into the system.

The VCMS Capacity Management System embodies the Knowledge Engineering approach; but before describing this we give a short account of its precurser DRAGON, the first ICL sizing system to be based on these principles.

Dragon

In the late summer of 1982, consideration was being given to producing a sizing system for use by ICL sales staff. Such a system would start by establishing the customer's future workload by asking a series of questions, adapting the questioning to the type of workload and the level of information available. Like the human expert, the system would need to change its line of questioning if one course was found to be unproductive. Where the user was uncertain of the answer to a question, the system would need to be able to give advice and make helpful suggestions based on what was already known – again like the human expert. The system would need to insist on certain minimum levels of information being provided and should then fill in the gaps where the user had been unable to answer less significant questions by taking knowledgeable defaults. Once the customer's workload had been defined, its computer resource requirements would be "costed" and this mapped onto a suitable hardware configuration. This would be modelled to ensure performance constraints – such as response time targets would be met. The system would need to produce a summary report for inclusion in sales proposals.

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

In October 1982 work started on an evaluation prototype – subsequently named Dragon[1,2], using the ADVISER expert system shell. To provide a fairly complete picture of the problems to be handled in an expert sizing system, it was decided that Dragon would need to take a vertical slice

through all the main steps in the sizing process, while at the same time restricting the range of products to be covered, to speed development.

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

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

The successful development of DRAGON showed that not only was such a system possible but also that the use of expert system techniques enabled a totally practical, robust, usable system to be created; and that this could reflect many of the attributes of the human expert, for example

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

For the beginner to knowledge engineering the use of the ADVISER package enabled knowledge to be encoded rapidly without the need to worry about data structures, user interfaces, etc. The very high level declarative language used to specify knowledge to ADVISER enabled rapid development to be made with few coding errors. ADVISER's ready acceptance by non-programmers made it possible for the expert system to be created directly by the person with the knowledge, rather than by working through an intermediary.

VME capacity management system

After considering the lessons learnt from DRAGON and in view of the report produced by the Performance Working Party of the Large Systems User Group[6], it was decided to develop a knowledge engineered capacity management tool as a product. This posed a particular challenge since the tool envisaged would provide the extraction, storage and reporting of relevant performance data as well as the more knowledge intensive analysis and modelling facilities. This tool, the VME Capacity Management System (VCMS), will now be described.

At its simplest, predictive sizing is no more complex than simple arithmetic and in fact such an approach would benefit the majority of computer

installations and would totally satisfy substantial numbers. In such an approach the units of resource demand are estimated for each unit of work in terms of OCP seconds to be used, number of transfers to each device which must be completed and the mainstore required. It is then necessary that the minimum numbers of each unit of work be processed, the minimum rate of processing each such unit and the required level of concurrency of processing each unit be identified for the periods to be sized. Simple calculations allow the total resource demands of the workload to be quickly quantified and these demands can be compared with the levels of demand which can be reasonably supported by the system while providing an acceptable level of response. This level of sizing is ideally suited to spreadsheet techniques and the use of simple resource usage data such as that provided by the VME Accounting and Budgeting Option and its use is recommended wherever mixed and volatile workloads exist or minimal resource is available for sizing.

Obviously once spreadsheet techniques are applied, additional sophistication is possible. For instance analysis of the sensitivity of the results to errors in frequency or rate of work unit processing can be undertaken and thus the efforts in determining the demand characteristics can be concentrated on particular work units. Furthermore the relative priorities of the different units or work can be taken into account so allowing higher loadings to be evaluated as safe when mixed workloads are being evaluated.

The approach can be further enhanced by increasing the level of detail in which the work units are specified thus allowing more of the configurational characteristics to be evaluated. For instance it would be possible to specify for each unit of work the number of transfers and of bytes transferred to each file. This taken with a mapping of file to disc, simple with good spreadsheet software, allows individual disc and controller loadings to be evaluated.

In fact, except for complex queuing models the use of spreadsheets to estimate network traffic and loadings is the only reasonable approach.

To exploit the use of spreadsheet techniques and to bring knowledge based systems to the support of configuring and workload planning decisions ICL has developed a system called VME Capacity Management System. This system, known as VCMS for short, provides facilities for constructing capacity plans for VME installations and monitoring actual usage trends against these plans.

The system is based on REVEAL a knowledge based modelling tool. REVEAL offers facilities for the development of models which may include rules and judgements as well as conventional algorithms. These facilities have been used by ICL to provide enhanced exception reporting and workload specification facilities. The VCMS system which is illustrated in Fig. 1 comprises the following components:

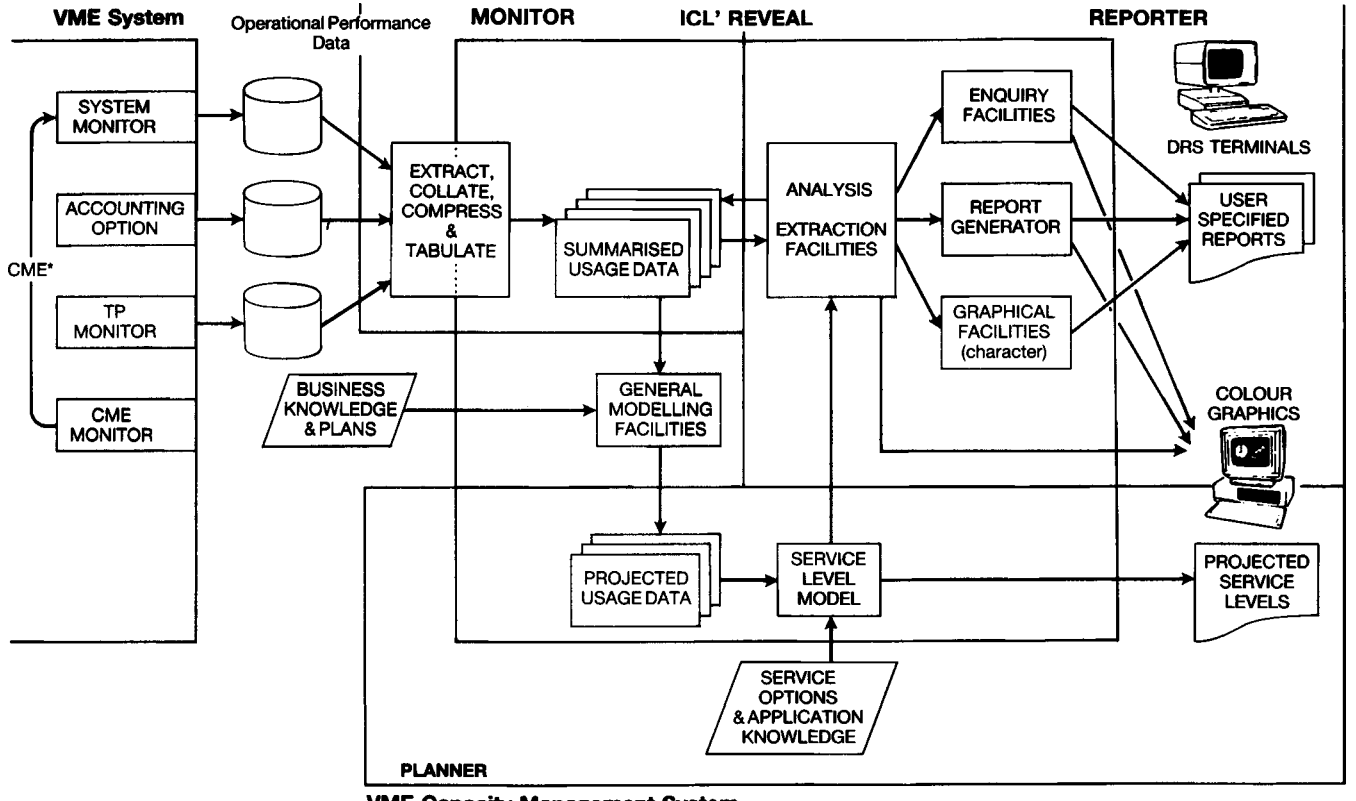


Fig. 1 VME capacity management system VCMS

VCMS MONITOR which extracts performance data from various sources in VME and builds up a performance database.

VCMS REPORTER which enables the user to produce his own reports and graphs from this performance database.

VCMS PLANNER which enables the user to build up capacity plans. This incorporates the lessons learned from the DRAGON system.

VCMS REVEAL which further extends the facilities available to include business planning and knowledge engineering.

Reveal and VCMS

The REVEAL system can be used to program conventional algorithmic operations in the normal way, using procedural programming techniques and the standard arithmetic, boolean and character string operations, loops and conditions. It enables complex problems to be modelled and it provides a wide variety of special functions including:

- data handling
- mathematical functions
- other special functions, such as the ability to operate on character strings.

The unique power of REVEAL, however, lies in its facilities for the representation of imprecise human knowledge. A REVEAL program, following the conventional procedural pattern, may also include a set of rules, written in a purely declarative form (without any particular order of execution being implied). Such a set of rules might, for example, define that:

if response.time is above about 5 then response.measure is unacceptable

if response.time is close to 5 then response.measure is quite acceptable

if response.time is well below about 5 then response.measure is very acceptable.

These rules include:

- linguistic variables (e.g. response.time and response.measure). Each linguistic variable has a name and a domain of applicability (e.g. from 1 to 100).
- qualifiers expressing a concept, for example "acceptable" or "about 5" which applies to a linguistic variable. Each qualifier is represented by a truth function, indicating the degree to which, for example, any response time in the range is considered to fit the concept. This representation is on a scale of numeric values from 0 to 1. These concepts can be further modified by hedges (e.g. "quite") which strengthen or weaken the force of the qualifier.
- noise words (e.g. our, should be, that of) to improve the readability of the statements without affecting their logical meanings.

When the model is executed the relevant rules are applied to the given set of input values.

Concept definitions

Concepts such as "maximum" and "acceptable" mentioned above are defined by the user for the particular purpose relevant at the time. These concepts once defined are retained in a vocabulary for future use. The definitions of the concepts are held as fuzzy sets and manipulated according to the rules of fuzzy logic. This simulates the process of normal human reasoning, in that it allows data to be viewed in terms of the various shades of possibility and uncertainty which almost always exist between the two poles of "true" and "false".

While conventional computer logic (and classical set theory, or two-value logic) allow for only two values (true and false), in reality human perception is able, by adding the factors of judgement, experience and expert knowledge to the situation, to realise that there is a range of values between true and false – the shades of grey between the black and white. Let us consider an example:

If the maximum acceptable response time for a service were considered to be 5 seconds, binary logic would accept a response of 4·99 seconds without qualification. Defining maximum acceptable response time using a fuzzy set allows grading to be performed so that acceptability may be qualified for times close to the maximum. This is illustrated in the graph shown below. This also illustrates how concepts are changed by the words "above" and "below".

VCMS knowledge base

Using the facilities provided by REVEAL expert knowledge has been

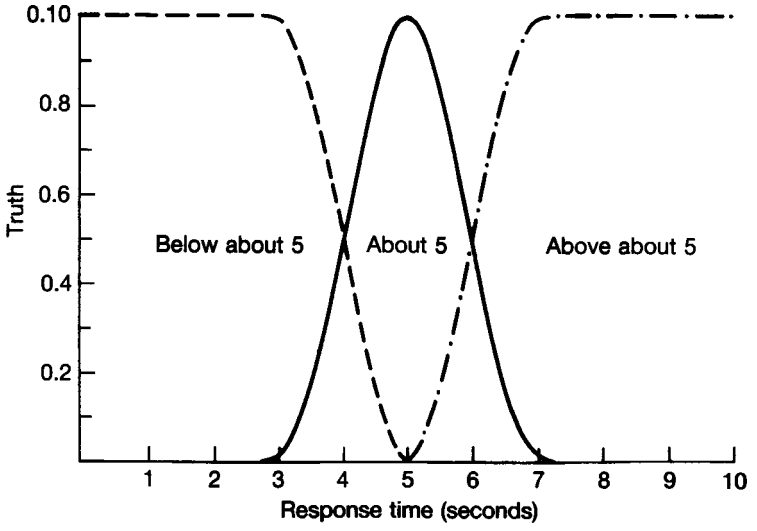


Fig. 2 "Fuzzy" concepts

incorporated into the VCMS Monitor, Reporter and Planner modules. The Monitor and Reporter contain a performance evaluation knowledge base which may be used to monitor the health of the system. The Planner contains knowledge of the structure of 2900 and S39 hardware and system software which may be used for modelling purposes.

The performance evaluation knowledge base consists of rules which may be used to analyse the monitored data taken in by the system. These rules check for abnormal or excessive response times, OCP, mainstore and disc usages, and disc unbalance.

The data concerning the processor utilisation by the different classes of work is analysed to check that the following are reasonable for the class of work:

total usage
interactive usage
TP usage
system usage
control usage.

The rules used are fairly complex taking into account amongst other things peaks and sampling periods.

The data concerning MAC usages are used to determine whether there is sufficient mainstore for the workload being process. The rules check for

store occupancy
excessive Roll in Roll out operations (RIROs)
excessive VSIs
VSI service times are reasonable.

The measured mainframe response times are compared with that which would be expected for the work being processed, assuming reasonable disc response times.

Data concerning the individual discs for which data has been collected is evaluated to check that:

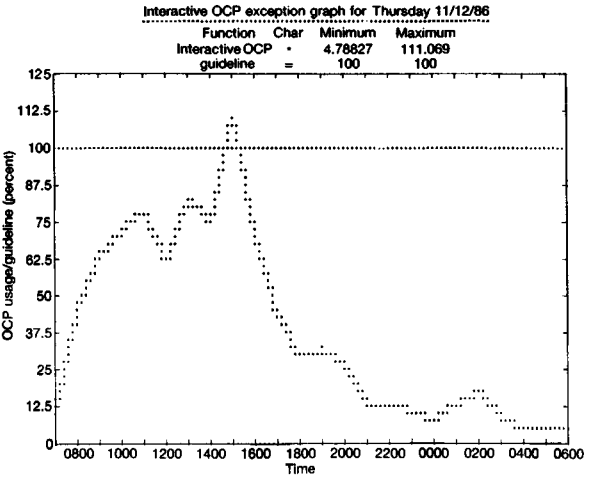
the transfer rate is reasonable
the transfer rate is not unduly above the average for the system
the transfer rate is consistent
user/Kernel transfers to the same disc are suitably balanced.

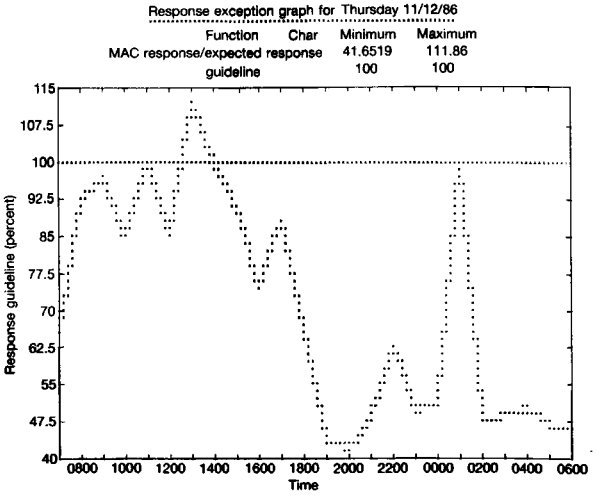
An example of a VCMS performance evaluation report is given in the Appendix: this reproduces *exactly* what is output by the system. Some of the graphs referred to in this output are given in Fig. 3.

VCMS Planner owes much to Dragon, in particular it includes the FAST (Football Analogy of System Throughput) network queuing model software

developed by Conway Berners Lee[3,4]. The module helps the user to construct and evaluate capacity plans for ICL 2900 and Series 39 systems, in two parts concerning workloads and configurations respectively. The first, workloads, represents the way in which the demands of different users or usages for the services of different types (TP, MAC, Batch) are planned to occur over the time period being considered. The second, configurations, represents possible ways of satisfying these demands. The two parts can be linked as required.

Workloads are held as tables in which demands are given against time, with a table for each use and each component of demand; these may be derived directly from VCMS Monitor data, as measured workloads, or input manually for new applications. Workloads may be aggregated so that combinations can be evaluated against a given configuration, thus enabling





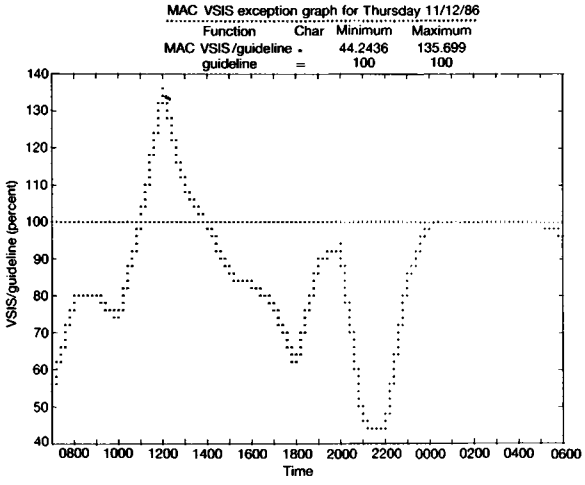


Fig. 3 VCMS performance evaluation report: examples of graphical output

the user to plan schedules (short term plans) and the phasing of projects (long term plans). Detailed evaluations giving breakdowns by individual components can be obtained, enabling the user to plan individual applications in the context of the overall capacity plan.

Configurations, which also may vary with time, are defined in terms of the names, labels or other identifiers used for ICL hardware equipment, and are input through form-type screens. Planner contains knowledge of the service characteristics of the relevant OCPs, discs, and disc controllers which is used to set up an appropriate network description for input to the queuing model and to perform conversions between different types of OCP. Knowledge of how VME system demands are incurred is used to include such demands as and when necessary.

Conclusions

This paper has described two expert systems for capacity planning. The nature of the problem demands a high degree of integration of solution methodologies and software technologies. These encompass:

- efficient processing and analysis of large volumes of computer performance data
- mathematical queuing theory models of system throughput
- heuristic knowledge of acceptable limits of demands applied to configurations
- transformation knowledge necessary to formulate real world questions into input to the queuing models.

The REVEAL System upon which VCMS is based has proved its value as a development tool for such integrated systems. The primary benefits of such a

tool are in the reduction of the development effort requirements and timescales. It has been estimated that, compared with conventional programming methods, REVEAL will reduce timescales by a factor of up to 4 for complete system development. Further benefit comes from the ability to build systems using REVEAL where the prototype is used as the specification. This enables an increase in quality of the final system produced in terms of usability and closeness of fit to user requirements.

Simple interfaces are provided to facilitate the inclusion of modules written using conventional language systems such as COBOL and the Data Dictionary System in systems developed using REVEAL. These were used to program efficient input of the volume performance data by VCMS Monitor. Also possible are links to modules written in the scientific languages FORTRAN and PASCAL. VCMS-Planner uses mathematical queuing software which already existed and had been written in PASCAL.

The REVEAL facilities for processing tabulated data according to rules as well as algorithms have enabled the inclusion into VCMS of the heuristic knowledge which is essential to the capacity management process.

It is expected that, by providing easy access to this knowledge, VCMS will prove useful to managers and technical support specialists responsible for ICL VME installations.

Acknowledgments

Acknowledgment is made to the many people who contributed to the development of the systems described in this paper. In particular M.J.R. Keen who initially developed DRAGON, and to R. Bayly for his contribution to VCMS.

References

1 KEEN, M.J.R.: "Dragon: the development of an expert sizing system". Eleventh European Conference on Computer Measurement, October 1983.
2 KEEN, M.J.R.: "Dragon: the development of an expert sizing system". ICL Technical Journal, 1983, **2** (4), 360–372.
3 BERNER-LEE, C.M.: "System Modelling – where we are and where we have to go". Seventeenth Annual Technical Symposium of ACM, 1987.
4 BERNER-LEE, C.M.: "Network models of system performance". ICL Technical Journal, 1979 **1** (2), 147–171.
5 BARTLETT, C.W.: "S39XC – A Configurer for Series 39 Mainframe Systems". ICL Technical Journal, 1987.
6 "Large Systems User Group". Report of the Performance Working Party, March 1985.

Appendix

Performance evaluation: System Monitor

Evaluation of System Monitor data for THU861211 found in file LATEST.

Processor subsystem

Evaluation of the usage of the processor subsystem has identified that the following guidelines are being exceeded:

Total OCP demand

The total OCP demands on the system are excessive for the type of work being processed during 1 hour. These guidelines are approximately:

95% utilisation over one hour for wholly BATCH work
67% utilisation for interactive work.

Exceeding these guidelines is likely to lead to a significant amount of queuing for the OCP. This will adversely affect the interactive response times and batch turnaround times.

The periods of the day when these overloads occur and their extent are shown in the associated graph entitled 'TOTAL OCP EXCEPTION GRAPH'. This graph plots total OCP, (charged and uncharged), as a percentage of the recommended limit for the mixture of work being processed. Thus the periods of day where this percentage is above 100% should be investigated.

Excessive interactive OCP demand

The demand for OCP by interactive work in total exceeds the recommended guideline of 67% during 1 hour. This is likely to lead to a significant amount of queuing for the OCP, particularly by the lower priority (MAC) work.

The periods of time and extent to which this guideline is exceeded is shown in the associated graph entitled 'INTERACTIVE OCP EXCEPTION REPORT'. This graph plots the OCP usage excluding batch as a percentage of the recommended limit. Thus the periods of the day where this percentage is above 100% should be investigated.

Excessive system OCP demand

The demand for OCP by system policies and the kernel is excessive considering other work being processed during 13 hours. This is likely to lead to poor response times for interactive work.

The periods of time and extent to which this guideline is exceeded is shown in the associated graph entitled 'SYSTEM OCP EXCEPTION REPORT'. This graph plots the OCP used by the system policies and the kernel as a percentage of the recommended limit. Thus the periods of day where this exceeds 100% should be investigated.

Excessive Policy 8 OCP demand

The demand for OCP by Policy 8 (Control) is excessive considering other work being processed during 2 hours.

The periods of time and extent to which this guideline is exceeded is shown in the associated graph entitled 'CONTROL OCP EXCEPTION REPORT'. This graph plots OCP used in Policy 8 as a percentage of the recommended limit. Thus the periods of day where this percentage is above 100% should be investigated.

Mac response analysis

Evaluation of the MAC response times determined from the System Monitor data for THU861211 loaded from file LATEST shows that the response times experienced were longer than would be expected for a well balanced system for 1 hour. The MAC response is related to ALL work in VME scheduling Policy 2. The response does not include delays due to communications systems.

The periods of the day when this occurs and the extent to which the response times are excessive are shown in the associated graph entitled 'MAC RESPONSE EXCEPTION GRAPH'. This graph plots measured response time as a percentage of expected response time. Thus the graph expresses response quality irrespective of the size of interactions being processed. If the measured response times are much greater than 100% of the expected times then the cause of this should be investigated.

Mainstore analysis

Evaluation of the usage of the mainstore subsystem has identified the following guidelines are being exceeded:

Excessive VSIs

The demand for VSIs by the MAC workload exceeds that which would be expected for the kind of work being undertaken for 2 hours. This is evidence that some adjustment to the workload would be beneficial.

The periods of the day when the overload is occurring and its extent are shown in the associated graph entitled 'MAC VSI EXCEPTION GRAPH'. This graph plots the VSIs per interaction as a percentage of the recommended limit. Thus the periods of the day where this percentage is above 100% should be investigated.

Disc subsystem

The data for individual discs found in the System Monitor data has been evaluated in respect of the following:

discs carrying an above average proportion of total traffic.
discs with very high transfer rates.
discs with high peak to average transfer rates.

Where results were obtained which could indicate any of the above a graph illustrates the periods of concern.

Volume	*Comment*
BACCAT	has a very high transfer rate and is potentially limiting.
BACCAT	has a very high peak to mean transfers.
ICL706	analysed OK.
ICLSS0	analysed OK.
ICLSS1	analysed OK.
ICLSS2	analysed OK.
ICLSS3	is a possible cause of poor VSI service times.
ICLSS4	analysed OK.
ICLSS5	analysed OK.

APPLICATIONS ENVIRONMENT

To deal with more complex problems that are arising, better tools are needed than have been available previously so as to make the resources of the system more easily available to the user. There has been a continuing effort over the whole history of computer use to reduce the amount of knowledge of the technicalities of the system that is needed for effective and efficient use of these resources.

On knowledge bases at ECRC

Jean-Marie Nicolas
ECRC*, Munich, FRG

Abstract

The purpose of this short note is to give a brief introduction to ECRC activity in the field of Knowledge Bases, while providing a summary of some of the software prototypes which have been developed so far in this context. A companion paper by Jorge Bocca presents in more details one of these, the Logic/Database programming system Educe.

1 On the context and the objectives

To quote a recent ICOT publication, "The 5th Generation computer is aimed at the realisation of Knowledge Information Processing ..." then software development in most areas will be viewed essentially as knowledge base construction. What else is needed to be said to indicate the importance of the knowledge base research with regard to the 5G project ...?

In such a context, the objective of the ECRC Knowledge Base Group is to contribute to the emergence and to the development of the technology on which will rely the construction of information management systems that will permit an intelligent and efficient manipulation of large knowledge or databases. The idea is to achieve systems which combine the high-level information modelling features, the deductive capabilities and the flexibility of AI-based systems together with the efficiency and the control facilities provided by database systems over large sets of information. It is worth emphasising that such systems have to be viewed not only as extended database management systems but also as knowledge programming systems to be used as high-level programming tools in the development of knowledge-based systems in, say, software engineering, business, diagnostic, medical or CAD applications ....

In order to achieve the above objectives, the ECRC-KB Group has chosen to analyse and to investigate, both on a theoretical and on a practical basis (i.e. via system prototypes construction), the various technical issues which are concerned with:

*The European Computer-Industry Research Centre is a joint Research Centre established by Bull, ICL and Siemens in January 1984.

- conceptual knowledge modelling, i.e. how to logically represent, structure, and organise knowledge in a powerful and natural way;
- physical knowledge representation, i.e. how to physically represent, structure, organise and access knowledge efficiently;
- extensions of information retrieval capabilities by effectively using deductive techniques either to enhance the user view of the knowledge base or to improve the responsiveness of the system;
- formal techniques to check for various properties of knowledge such as validity of factual knowledge with regard to integrity constraints or consistency of rules, so as to obtain more secure systems.

2 Approach and current software prototypes

Our approach to the development of Knowledge Base Systems, as introduced above, aims at taking advantage, as much as possible, of a smooth integration of Artificial Intelligence and database technologies, along the lines which have been drawn up by research on deductive databases[9]. This approach to KBMS building seems to become nowadays the common rule (e.g. see [5]); it is not fortuitous. Indeed, it is in the AI world that one finds the inference mechanisms that provide the bases for an intelligent manipulation and control of information and it is in the DB world that one finds efficient and secure management techniques for large sets of information[4].

One of the key issues in the development of such knowledge base systems is that of defining a harmonious (efficient) cooperation between inference techniques and database querying techniques so that deductive manipulation of large knowledge bases, stored on secondary devices, becomes computationally tractable. We have addressed this problem by taking two different but complementary approaches, one investigating various kinds of connections which can be set up between two given systems, the other aiming at the extension of the query/answering component of a DBMS, with a specific deductive unit, to support (recursive) derived relations (or views).

Our work on the first approach has culminated in the realisation of a system termed Educe* [2,3], which offers, in an integrated way, both a loose coupling and a tight integration between the inference system (and programming language) Prolog and the relational DBMS Ingres. Educe is certainly not the first Prolog/DBMS interface which has been developed; but, to our knowledge, it is the first to support both clauses and facts on disc, and to fully and efficiently handle Prolog recursive clauses. It is worth noting that depending on the emphasis which is put on either one of these two connections, Educe can be viewed and used in two different ways. It is a DBMS extension in the sense that it provides the database application programmer with Prolog as host language for the DBMS query language. It is a logic/database programming system in the sense that it permits, say, the knowledge engineer to write huge Prolog programmes whose clauses are efficiently managed on secondary devices.

*See the paper by Jorge Bocca in the same issue.

The prototype which has been realised in the context of the second approach is a deductive query/answering database system developed, via Educe, as an extension to a relational DBMS. According to a more conventional database terminology, this system, termed DEDGIN[11], can be referred to as a "view" (derived relation) handling mechanism which supports recursive "view" definitions. DEDGIN is based on a newly designed algorithm (QSQ)[12,13] which, for any kind of recursion, guarantees termination of the query evaluation process, ensures answer completeness and restricts access to those data which are effectively needed to answer the query. If various methods for recursive query evaluation have been recently proposed (e.g. see [1] for an overview), only a few of them are general in the sense that they offer the above mentioned properties for any kind of recursion and of database. According to [1], QSQ is the general method which offers, on average, the best performances. As a last point, because the view definition language which has been retained in DEDGIN is based, like Prolog, on Horn clauses, DEDGIN also appears as an "optimiser" for recursive clause handling. As such, it is at present being integrated in a new version of Educe in order to provide an alternative evaluator for (unordered) recursive clauses.

Efficient deductive manipulation of large sets of knowledge is one but not the single key issue in building KBMSs. Powerful information modelling features and efficient semantic integrity enforcement mechanisms are also needed. In order to tackle these problems, an Educe-based KB system has been realised. This system, called KB2[14], offers a semantic information model which is broadly an Entity-Relationship model extended with the "generalisation" logical structuring feature (IS-A hierarchy in the AI parlance) and its integrity subsystem, termed SOUNDCHECK, implements an efficient constraint enforcement technique we have defined[7] which reduces as much as possible the set of information to be checked for controlling update validity. An essential characteristic of this system lies in its versatility and flexibility of use provided by adequate schema manipulation facilities and by a smooth interconnection between the implementation language (Prolog/Educe), the (logic based) query and rule definition languages, and the knowledge manipulation language (Prolog again).

3 Conclusion

The three system prototypes we have introduced above certainly represent significant steps forward in the integration of AI and DB technologies for the development of KBMSs. However, if each of them is a useful system per se, none of them (even KB2) constitutes a full fledged KBMS. They are components of such a system and other functionalities need to be introduced. Thus, the development of these prototypes has been complemented with other studies which either focus on new functionalities (e.g. consistency checking for rules and constraints[6]), or on techniques to improve the efficiency (e.g. specific file organisation for KBs[8], negation by constraint[15]) or on more basic problems such as query compilation[10]. The results of these studies should give rise to extended versions of the systems.

References

1 BANCILHON, F., RAMAKRISHNAN, R.: 'An amateur's introduction to recursive query processing strategies'. In: Proc. ACM-SIGMOD conference. May, 1986. Held in Washington DC, USA.

2 BOCCA, J.: 'On the Evaluation Strategy of EDUCE'. In: Proceedings of the ACM-SIGMOD Int. Conf. on the Management of Datas. May 28–30, 1986. Washington DC, USA.

3 BOCCA, J.: 'EDUCE – A Marriage of Convenience: Prolog and a Relational DBMS'. In: Proceedings of the 3rd Symp. on Logic Programming. Sept., 1986. Salt Lake City, USA.

4 BOCCA, J., DECKER, H., NICOLAS, J.-M., VIEILLE, L. and WALLACE, M.: 'Some Steps towards a DBMS based KBMS'. In: Proceedings of the 10th World Computer Congress IFIP. Sept., 1986. Dublin, Ireland.

5 BRODIE, M., MYLOPOULOS, J., eds.: 'On Knowledge Base Management Systems, Integrating Artificial Intelligence and Database Technologies'. In: Springer Verlag. 1986. New York, USA.

6 BRY, F. and MANTHEY, R.: 'Checking Consistency of Database Constraints: a Logical Basis'. In: Proceedings of the 12th Int. Conf. on Very Large Data Bases. August 25–28, 1986. Kyoto, Japan.

7 DECKER, H.: 'Integrity Enforcement on Deductive Databases'. In: Proceedings of the 1st Int. Conf. on Expert Database Systems. April 1–4, 1986. Charleston, South-Carolina, USA.

8 FREESTON, M.: 'Data Structures for Knowledge Bases: multi-dimensional file organisations'. In: ECRC TR-KB-13. August, 1986, partly published under the title "BANG file: a new kind of grid file" in Proc. ACM-SIGMOD Conference, May 27–29, 1987, San Francisco, USA.

9 GALLAIRE, H. MINKER, J. and NICOLAS, J.-M.: 'Logic and Databases: a deductive approach'. In: ACM Computing Surveys. June, 1984.

10 DE ROUGEMONT, M.: 'Theory and practice of intentional compilation of queries'. In: ECRC IR-KB-22. 1986.

11 VIEILLE, L.: 'Recursion in Deductive Databases: DEDGIN, a recursive query evaluator'. In: ECRC TR-KB-14. November, 1986.

12 VIEILLE, L.: 'Recursive Axioms in Deductive Databases: The Query-Subquery Approach'. In: Proceedings of the 1st Int. Conf. on Expert Database Systems. April 1–4, 1986. Charleston, South-Carolina, USA.

13 VIELLE, L.: 'Recursion in Deductive Databases: a db-complete proof procedure based on SLD-Resolution'. In: ECRC TR-KB-15. November, 1986, also published in Proc. 4th Int. Conf. on Logic Programming, May 25–29, 1987, Melbourne, Australia.

14 WALLACE, M.: 'KB2: a Knowledge Base System embedded in Prolog'. In: ECRC TR-KB-12. August, 1986.

15 WALLACE, M.: 'Negation by Constraints'. In: ECRC IR-KB-25. June, 1986.

Logic languages and relational databases: the design and implementation of Educe

Jorge Bocca
European Computer Industry Research Centre, Munich

Abstract

The design and implementation of a logic programming system capable of handling large knowledge bases is presented. This system, known as Educe, has been constructed by fully integrating the logic programming language Prolog and the relational database management system Ingres.

1 Introduction

Recently, there has been an upsurge of interest in the techniques for the construction of large Knowledge Base Management Systems (KBMS)[9,10]. A number of proposals have been made in order to implement a logic programming system capable of handling large knowledge bases. M. Stonebraker groups these proposals into three categories[16].

The first case consists of the integration of a suitable file system into a rule manager such as a Prolog interpreter. Examples of this approach are K.U.L. Prolog[19] and Mu-Prolog[13]. A second approach is to extend a database management system (DBMS) to support rules and inference[2,16]. The last case, the full integration of a DBMS and a rule manager such as Prolog, is generally acknowledged as the best possible solution of the three cases[16,18]. The main objection to a logic system constructed on these lines has been the predicted difficulty of its implementation. However, this objection has not halted the development of some experimental systems based on the integration of Prolog and a DBMS[2,6,8,19,18]. Although these developments have produced significant progress, an acceptable logic programming system with the required capabilities has not previously been achieved[18,22]. In our opinion this is due to the very weak form of integration of the deductive and the external database (EDB) components of the proposed systems[9].

Initially, Educe used a coupling between a Prolog interpreter and a relational DBMS for the implementation of the loose DML. Because of the problems

that the evaluation of recursive queries causes[3] in a coupled system, the close DML was implemented by integrating the low level access mechanism of the DBMS into the Prolog interpreter. Although these two approaches might be thought antagonistic to each other, in Educe they co-exist and co-operate.

At the top level, Educe offers users two different languages: one following the non-procedural style of data manipulation language (DML) for relational database management systems (RDBMS) and one with a style close to Prolog. We refer to these languages as *loose* DML and *close* DML, respectively. Expressions in these languages can be freely mixed in Educe programs. In terms of implementation, there is a close correlation between these languages and the evaluation strategies outlined in the previous paragraph. It seems natural to use the *sets retrieval* strategy for the loose DML and the *one-tuple-at-a-time* strategy for the close DML.

This paper attempts to bring together the main ideas and techniques used in the design and implementation of Educe. Some of these ideas have been discussed in detail somewhere else and adequate references are given, throughout the paper.

This paper is divided into seven sections. Section 1 is this introduction. In section 2, the concepts for a logic programming system for KBMS are examined. Section 3 describes Educe's user interface and the reasoning behind it. Sections 4 and 5 deal with the general design and the particular implementation techniques used in the construction of Educe. Section 6 discusses significant problems found in the development of Educe and presents the adopted solution (if any). Conclusions are presented in section 7.

2 The solution's levels

The integration of Prolog and a DBMS, as the deductive and the external database (EDB) components, respectively, could have taken a number of forms. In order to discuss them, we separate logical issues from the physical implementation itself. At the physical level, it is important to distinguish whether the logic system is the result of a coupling of the two components or a proper integration. In the case of a coupling, the individual components remain as two independent processes with a capability to communicate with each other. In the case of integration, the two components become one unit with a blurred interface. The logical level is concerned with the user's interface and its effects on the overall design of the logic system.

Depending on the degree of physical integration of the two components, we use the terms, **coupling** and **integration**, to refer to a rather loose coupling of the two components or to the convergence of the two components into one fully integrated system, respectively. At the logical level and based on the degree of *mimetisation* with Prolog of the data manipulation language (DML), we again distinguish two cases: **close** and **loose**. The former term

refers to a DML that follows the notation and conventions used by Prolog, while the latter designates a DML with syntax and conventions similar to DMLs in relational DBMSs. In what follows, we also use the term **coupling** in a generic sense, i.e. whenever the context makes it unnecessary to specify whether the physical interface between the components is a coupling or integration.

2.1 The logical level: loose vs. close

Perhaps the most obvious advantage of a close DML to the Prolog programmer is its complete transparency. This transparency makes a close DML a very attractive choice. In particular, prototype systems can be implemented in the first instance without a DBMS facility, and at a later stage, when the amount of data exceeds certain limits, the DBMS facility can be added. This should not cause changes to the programs on account of the DBMS addition.

Unfortunately, the advantages of close coupling are not exempt from some serious drawbacks. Consider the following definition of *p* and the base relations *employee*, *q* and *r*:

```
p( Name, Salary, Other ) :-     /* (1) */
        employee( Name, Salary ) ,
        funny( Salary, Other ) ,
        Salary = 100 .

employee(a, 20) .
employee(a, 100) .
employee(b, 100) .
        :

q(10, xx) .
q(100, xx) .
q(100, yy) .
r(xx, aa) .
r(yy, bb) .
```

In the evaluation of ?- *p(Name, ...)*, the use of definition *(1)* determines that every tuple in *employee* has to be retrieved and tested. Even to satisfy this goal once, a number of tuples proportional to the cardinality of *employee* would have to be retrieved. Worse, if there were no employee with *salary = 100*, all the tuples in *employee* would still be retrieved and tested. In order to satisfy the goal *p* more efficiently, it is tempting to "optimise" the body of *p*, by its logical transformation into the "*equivalent*":

```
p( Name, Salary, Other ) :-     /* (2) */
        employee( Name, 100 ) ,
        funny( Salary, Other ) ,
        Salary = 100 .
```

which unfortunately is **not** equivalent if *funny* was defined by:

```
funny( S, O) :-
        q(S, A) ,
        ! ,
        r(A, O) .
```

Clearly, to a Prolog programmer *(1)* and *(2)* are not equivalent. But to a user thinking in a "non-procedural" way, e.g. a user of a relational DBMS, *(1)* and *(2)* appear to be equivalent forms of expressing a query.

The alternative, a loose DML, can be seen as the embedding of a high level non-procedural DML (QUEL[15]) into a logic programming language (Prolog). In this case the atomic unit of access between control statements is the relation and not individual tuples. Hence problems such as *(1)* above never arise. The programmer is forced to express explicitly what he/she wants. For instance, in the case of rule *(1)*, the programmer would have to write the rule unambiguously as either:

```
p( Name, Salary, Other ) :-     /*(1') */
        retrieve(   [employee.name = Name ,
                     employee.salary = Salary] ,
                    employee.salary = 100
                ) ,
        funny(...) .
```

or

```
p( Name, Salary, Other ) :-     /* (1") */
        retrieve(   [employee.name = Name ,
                     employee.salary = Salary] .
                    true /* all qualify */
                ) ,
        funny(...) ,
        Salary = 100.
```

2.2 The physical level: coupling vs. integration

At first it appeared that physical coupling of Prolog and the EDB could produce better and faster solutions than integration. A coupling could quickly and easily be implemented by running two intercommunicating processes: one for Prolog and one for the EDB. This sort of coupling could also greatly benefit from the performance advantages of data base machines.

However, it should be noticed that a solution at the physical level is not independent of the decisions made at the logical level. For instance, take a close coupling of Prolog and an EDB, and then consider a query on equality over the relation *employee* which is stored as a *hash* file. In this case, the correct strategy would be to use the hash function to get the right tuple(s) instead of searching the file sequentially. By the same token, if the relation *employee* was stored using index sequential access method (ISAM) on *salary*, and *highpaid* was defined by:

```
highpaid(Name) :-      /* (3) */
        employee(Name, Salary) ,
        ..., Salary > 100.
```

then to evaluate the goal: *?- highpaid(N).* one would like to make use of the ISAM structure of *employee.* Unfortunately, because of the problem with control predicates, one would be "forced" to retrieve sequentially the relation *employee.*

To avoid these situations, we extended the syntax of Prolog (close). Thus, the above definition can be written in Educe, as:

```
highpaid(Name) :-      /* (3') */
        employee(Name, Salary > 100) ,
        ....
```

To further illustrate this interplay between the logical and the physical levels, consider once more a loose coupling. For instance, some versions of Mu-Prolog[13]. In these implementations whenever a relation is opened a new process and two pipes are set up. In terms of the operating system's resources, this slows the overall response time of the local UNIX[17] system by consuming a considerable amount of its limited resources. Gains in time, obtained by a very efficient access mechanism in Mu-Prolog[14] are thus almost completely lost in the communications between the processes.

2.3 Some related work

Couplings of a close type have reached a significant level of refinement. In addition to the work of L. Naish and J.A. Thom in Mu-Prolog, another interesting but partial implementation of close coupling is the one by Vassiliou et al.[18]. This implementation assumes that answers returned by the DBMS are always small enough to be kept as assertions in main memory. This approach probably works very well in applications with a relatively small number of facts, but in the general case it gives rise to some questions. For instance, in recursive searches often all the tuples of a relation are retrieved. Obviously, the main memory restriction limits the size of the databases.

As for loose coupling, a few systems have been reported which are capable of converting a Prolog goal into an equivalent query to a database management system (DBMS)[6,11,1]. The main restriction in nearly all of these systems is that they are not capable of handling recursion on base relations.

More recently, Faget et al.[8] reported the implementation of Frog. Although Frog does not attempt a full integration of Prolog and a DBMS, its overall design meets almost the same problems. In Frog, a number of the problems discussed above have been solved but issues of control are avoided.

Unfortunately, as far as we are aware, Frog fails to provide general recursion on relations, despite the fact that the syntax allows it.

Very few attempts have been made to integrate (as oppose to couple) Prolog and a DBMS. Except for current work on a newer version of Mu-Prolog[12], work on integration has been restricted to proposals to add deductive features to existing DBMS[2,10].

More significatively most of the work reported so far only deals with querying a DBMS in single user mode. General DBMS facilities such as concurrent queries/updates are not supported. Of course, the proposals to extend existing relational DBMS[2,16] are exempt from this restriction.

As for systems which have attempted integration, they hardly exist[8,19,18]. Hence the difficulty in drawing conclusions in this respect.

3 A user's view

From the user's point of view, Educe can be seen as an extended Prolog system. Although most of the extra features are transparent to users, a number of built-in predicates had to be added to explicitly manipulate the underlying DBMS. Facts and rules can be stored in the EDB. Facts are stored in base relations according to their semantic while all rules are stored in one relation: *rulerel*. Rules can be added, updated, retrieved, etc. just like the facts stored in ordinary relations. Once a rule is stored in the EDB, it can be used like any other rule. No explicit syntax is required to access and/or execute a rule stored in the EDB. The access to facts stored in the EDB can also be made transparent to users (close access). However, if users so prefer, facts (and rules) in the EDB can be retrieved by using the *retrieve* predicate. This predicate uses non-positional expressions to refer to relations, attributes and retrieval conditions (loose access). To convey the flavour of Educe's user interface, a small sample of built-in predicates is discussed in this section. For a full user's manual, see [7].

A first group of built-in predicates allows users to set up, destroy and connect to databases. An example in this group is the predicate *db*:

```
?- db(refs) .
```

to activate the database *refs*.

A second group of built-in predicates provides information about the catalogues of the database. The predicate *helpr* is typical of this group. The call

```
?- helpr( paper ) .
```

prints details about the relation *paper*. The information printed includes the names of the attributes, their types, storage structure, etc.

Then there is the group of predicates to manipulate the database schema, i.e. to create new relations, destroy existing ones, etc. An example in this group is the predicate *create*. For instance,

```
?- create(
        paper( number = i2, rating = i1,
        author = c20, title = c30,
        source = c10
        )
).
```

creates the relation *paper* with attributes *number, rating, author, title* and *source.*

Predicates to add, delete and update facts are also provided. A boolean condition is used to specify the set of tuples to be deleted or updated. An example of this is the predicate *erase*:

```
?- erase( paper,
        (paper.rating = 0) or
        (paper.source = unknown)
  ).
```

deletes all tuples of relation *paper* such that the *rating* is 0 or the *source* is unknown.

Three forms of retrievals are presented below. The *loose* DML retrieval is exemplified with the query:

```
?- retrieve([ paper.number = N,
              paper.rating = R] ,
      (paper.author = 'Brown, John')
      and (paper.rating > Min)) ,
   Min = 5.
```

which produces:

```
N = 5
R = 10
Min = 5 ;

N = 9
R = 7
Min = 5
```

i.e. it retrieves the *paper.number* and the *paper.rating* of papers with a *paper.rating* higher than 5 that *Brown, John* has written. Notice that Prolog variables **can be used** in the Condition part of the retrieve. Answers are obtained by the instantiation of variables in the Atts part of *retrieve.* This mechanism allows for the definition of (virtual) relations which behave just like a Prolog fact.

Another form of a *loose* retrieval is provided by the predicate *save*. This is a set retrieval which facilitates the storage of results (ResRel). An example of its use is

```
?- save( [employee] ,
        highpaid( name = employee.name ,
                  sal = employee.sal) ,
                  employee.sal > 4000 ) .
```

to silently retrieve all the employees (from relation *employee*) who earn more than 4000 and store their names and salaries in the relation *highpaid*.

The built-in predicate *retr(Rel)* is the primitive form for retrievals based on close access to the DBMS. The format, order and number of attributes in relation *Rel* must correspond exactly to the physical relation in the database. Transparency of access is achieved by the definition of a rule which has only the call to *retr* in its body. For example, to successfully evaluate

```
?- anc( X, charles ) .
```

anc should be defined by

```
/* interface to the physical relation */
parent( X, Y ) :-
        retr( parent(X,Y) ) .

/* recursive call */
anc( X, Y ) :-
        parent( X, Y ) .
anc( X, Y ) :-
        parent( X, Z ) , anc( Z, Y ) .
```

To make the use of *retr* implicit would cause a number of problems. This is discussed later on.

For large relations, the use of indexes is strongly recommended to improve performance. For this, facilities to create and maintain indexes have been provided in the Educe system. An instance of this kind of predicate is *index*. Its use is illustrated with the call

```
?- index( x2parent, parent, [father] ) .
```

which creates the secondary index *x2parent* for the relation *parent* on attribute *father*.

Also in Educe, a large number of miscellaneous predicates are provided. A couple of examples to demonstrate the facilities provided by the predicates in this group are

```
?- attributes(paper, Atts) .
```

which produces the list

```
Atts = [number, rating, author,
        title, source]
```

and the goal

```
?- ftor('/usr/mark/ppp', like) .
```

to copy the *like* facts in file */usr/mark/ppp* to the already existing relation *like*. As already stated, rules can be stored in the EDB. A number of predicates exist for the manipulation of these rules. It should be noticed that, due to the ordering contradiction between Prolog and the relational model, users are asked to specify the order in which rules should be consulted. Prolog has an implicit order of evaluation, from top to bottom, while tuples in a relation (and hence rules in *rulerel*) by definition are not ordered. Also, because of the potentially large numbers of rules in a definition, more selective predicates have been added for listing rules in the EDB. A couple of examples to illustrate the use of these predicates are given below,

```
?- nrule( 1,
    'anc(X,Y) :- parent(X,Y) .') .
?- nrule( 2,
    'anc(X,Y) :- parent(X,Z), anc(Z,Y) .') .
```

to store the two rules for *anc(X,Y)* in the external database. To evaluate the predicate *anc(A,B)*, Educe will try first the non-recursive rule (rule-1) and then the recursive rule (rule-2);

```
?- irule(anc) .
```

prints the two rules and their order of evaluation,

```
1. anc(X,Y) :-
        parent(X,Y) .
2. anc(X,Y) :-
        parent(X,Z) ,
        anc(Z,Y) .
```

4 Architecture

In this section, an outline of the chosen architecture for Educe and the motivations behind it are presented. A detailed discussion of possible alternative architectures for Educe can be found in [3].

From the point of view of the implementor, loose coupling presents itself as an obvious method for implementation. Provided that recursion is not allowed, a simple way to construct a loosely coupled system is by setting up two processes: one for the deductive component and one for the EDB

component. These two processes exchange messages, i.e. queries and replies, through a channel of communication. Educe follows this approach for loose coupling, setting up one process for Prolog, as the deductive component, and one process for Ingres as the EDB component. Communication between the Prolog and the Ingres processes is by means of two pipes[17], one for queries and one for replies.

Unfortunately, the two processes in loose coupling would be very inefficient for an implementation of the close DML in Educe. This is apparent in systems that have adopted this as a solution[8,13]. Because of this, we chose to integrate the deductive and the EDB components into one monolithic unit to handle the close DML. For this, the *access methods* module of the DBMS was detached from it and attached to Prolog.

This allowed the multiple process configuration of loose coupling to be merged with the close integration configuration in a particularly coherent way. To explain this, let us start by considering two concurrent processes, each of which runs the DBMS on a common database.

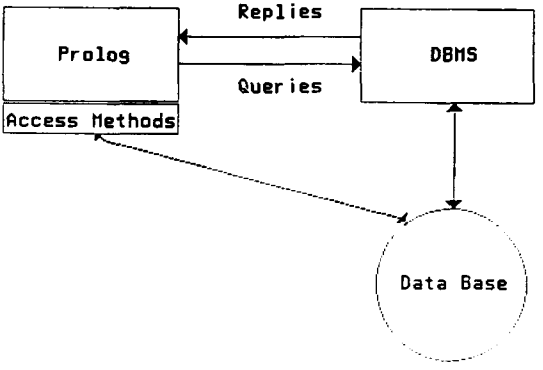


Fig. 1

When this last configuration is merged with the two previous ones, it produces Educe's architecture (Fig. 1). In the two DBMS configurations one of the occurrences of the DBMS is replaced by the Prolog + AM configuration. This is possible, since the Access Methods module of the Prolog + AM and the DBMS are identical replicas. In other words, Educe appears as two concurrent DBMSs sharing access to a common database.

It is important to note that this architecture does not impose any restrictions on recursion. On the contrary, it provides an efficient mechanism for the evaluation of multiple and recursive queries in either of the two languages, close and loose. Recursive definitions which include expressions in loose form are evaluated by a hybrid strategy. An evaluator has been implemented for this purpose. The evaluator uses loose coupling for the non-recursive part of the definition, and then, for the recursive part, it uses the route provided by

close integration for retrievals from the intermediate results. Recently, a module which performs mappings from expressions in loose form into close form and vice versa, has been built. This module allows Educe to select a route, either coupling or integration, entirely on the basis of expected performance.

5 Implementation

The description of the implementation of Educe here presented follows the historical development of the system. In order to avoid dismantling the deductive and the EDB components, loose coupling was implemented in the first instance. The integration phase was postponed until we had gathered sufficient detail of the construction of these components. Rule storage and the transformation of expressions can be seen as important extensions of the basic capabilities of Educe.

5.1 Loose coupling

Let us begin with the discussion of a relatively simple part of the implementation: the part that deals with loose coupling without recursion. This part of Educe was implemented as two related processes. One process acting as a master runs the Prolog interpreter, while the second process, the server, runs the DBMS. The Prolog interpreter used is a derived product of the Mu-Prolog interpreter[13]. The Ingres DBMS[15] was chosen as the EDB component. Thus, in this set-up, whenever the evaluation of a goal requires access to the EDB, all expressions requiring some form of syntactic analysis are parsed, and code is generated for them by the Prolog interpreter. The code generated is the equivalent QUEL expression. This QUEL expression, the query, is sent via a pipe to Ingres. Ingres in turn evaluates the query and produces a reply. This reply is piped back to Prolog which further processes it to bind variables to their respective values. In this part of the implementation, the control of processes and communication between them was written in C, while the parsing of queries and code generation was all done in Prolog. This scheme permits a more refined and efficient control of synchronisation and communication between the processes.

The predicate *helpdb* is perhaps the simplest example of the theory of operation described above. This predicate is defined by the Prolog clause:

```
helpdb :-
        query(' help ') .
```

The predicate *query(anIngresQuery)* takes the atom *anIngresQuery* and sends the string of characters forming the name of the atom down a pipe to Ingres. Then it waits for the evaluation of the query by Ingres and on completion returns *true*. Thus, by means of this mechanism, any arbitrary Ingres query can be sent to the server for evaluation. The predicate *query* was written in C and has been integrated into the Prolog interpreter.

A more complex situation develops when the mode of operation of Prolog differs from the mode of operation of the EDB. Take the case of *retrieve*:

```
retrieve( Atts, Boolean ) :-
            :
            :
   send_query( Rels, Atts, Boolean ) ,
            :
            :
   repeat,
   rel( P ) ,
   ( P = E,
      ( E = continue,
        !,
        fail
      )
      :
      :
      ;
      P = .. [ | OutAtts],
          value( Atts, OutAtts )
   ) .
```

Following the parsing of Atts, a query is sent to Ingres via *send query*. Typically, Ingres produces a whole relation as an answer to the query. Since Prolog requires only one tuple at a time, some adjustments have to be made. Basically, Ingres pipes the result relation to Prolog while Prolog takes one tuple each time from the pipe. In other words, the pipe acts as a queue. To take one tuple from the pipe, *rel(P)* is called. *P* is compared against the atom *continue* to check for the end of reply from Ingres. If it is not the end of reply (*continue*), then the tuple *P* is passed in suitable form to *value* which binds variables to attribute values.

It should also be mentioned that the syntax of *retrieve* allows uninstantiated variables in the conditions (*Boolean*) argument. Because of this, it is desirable to delay evaluation of the *retrieve* until the search criteria have been clearly established, so avoiding retrieval of unnecessary data.

5.2 Close integration

It is not only for syntactic convenience that variables are necessary in the condition part of *retrieves*. Without them, it would be impossible to express recursion. Take for example the relation *parent(X, Y)*, defined by:

```
parent( X, Y) :-
   ( var(X), var(Y), !,
     retrieve([ father.is_ = X,
        father.of_ = Y] ,
     true)
   )
   ;
   ( atom(X), atom(Y), !,
```

```
        :
    ) .
parent( X, Y) :-
        :
        retrieve([ mother.is_ = X,
            mother.of_ = Y] ,
        true)
          :
        .
```

Then we could define the derived relation *ancestor(X, Y)* recursively by:

```
ancestor( X, Y) :-
    parent( X, Y) .
ancestor( X, Y) :-
    parent( X, Z) ,
    ancestor(Z, Y) .
```

Unfortunately, the introduction of recursion and multiple queries brings new problems. It becomes necessary to relate replies to their originating queries. In order to evaluate a recursive definition such as *ancestor*, Educe first generates the relation *parent* (if virtual) and then proceeds to evaluate the recursive clause by using close access. But, before we discuss the details of how this is done in Educe, a description of the implementation of *close integration* is needed. The particular reasons for having close integration in Educe are discussed in [3].

Because of its linkage to the low level *access methods*, the close DML is implemented mainly in C. At the top level, the Prolog predicate *retr* binds the C implemented parts together. The implementation of this predicate is presented below:

```
retr( R ) :-
   R = .. [Rel | Attsl] ,
   openr(D, O, Rel) ,
   setsearch( D, Atts) ,
   repeat,
   getvals(D, Atts, NewTuple) ,
   ( ( /* failed */
      NewTuple = O ,
      closer(D),
      !, fail
    )
    ;
    true
   ) .
```

An example of the use of *retr* is *?- retr(employee(john, Salary))*. In this example, the relation employee is searched for *john's Salary*.

The programe above starts by transforming the (only) argument of *retr* into a list. The head of the list is instantiated to the name of the relation (*Rel*) and the tail is instantiated to a list of attributes (*Atts*), some as variables and the

others as constants, according to the particular retrieval condition. In our example, this list becomes [*employee, john, Salary*]. Once this is done, the relation *Rel* is opened by *openr* and the searching keys are set by *setsearch*. Only then the first tuple (if any exists) is retrieved by a call to *getvals*. The call to *repeat* is necessary to handle backtracking. The predicates *openr*, *setsearch*, *getvals* and *closer* are all implemented in C. The call to *openr* opens the file which contains the relation *Rel* and it also creates a descriptor, *D*. If the file for relation *Rel* was already open (for reading) then only the new descriptor D is created.

Descriptors not only keep static information about a relation, e.g. file name, degree, cardinality, etc., but they also maintain information of a dynamic type. In particular, information about the last tuple accessed is kept by the descriptors (TID of last tuple[15]). This is essential for an efficient implementation of backtracking. Otherwise, Educe would need to re-access old tuples to get the next tuple. This use of descriptors is essential in recursive cases. Without the descriptors, recursive queries on a given relation would be restricted to as many levels of recursion as the numbers of files that the host operating system allows to keep opened at any particular time. As an additional bonus to the scheme of operation described here, the overhead of opening and closing files is greatly reduced. In fact, for recursive queries, this overhead is reduced to practically nothing.

5.3 Rules

Once the ability to handle large numbers of facts by the EDB component had been installed in Educe, the next logical step was to introduce facilities to store and maintain large numbers of deduction rules in the EDB. Thus a mechanism to serve this purpose was implemented. In Educe, deduction rules are stored like the schema of the database, in a relation. This relation is named *rulerel*.

Obviously, the storage of rules in the EDB is not in itself enough to achieve the desired effect. Rules stored in the EDB must also be executed just like any other rule in main memory. For this, the top level Prolog interpreter was modified. Thus if, during the evaluation of a goal, an appropriate clause head is not found for it in main memory, then the *rulerel* relation is searched for it. If a rule with such a head is found in the EDB, then Educe executes it. If however the rule is not found in *rulerel* then Educe looks for a base relation to match the goal. If such a relation is found then the rule

```
Relation :-
   retr( Relation) .
```

is asserted. More formally, to evaluate a given goal G the algorithm is:

1 Search for rule/fact in main memory.

2 If 1 fails then search for rule in relation *rulerel.*
3 If 2 fails then search for base relation with matching name and degree. If such a relation is found then assert the rule *G :- retr(G).*

The algorithm above effectively makes the EDB component of Educe transparent to users of the close DML.

Unfortunately, this scheme of operation is not free of some (minor) side effects. The above algorithm implies an order of evaluation for rules and facts. In this implied order of evaluation, rules precede facts. However, since facts are a special case of rules (no body), they can be treated like a rule if so wanted by the user.

Still on the subject of evaluation precedence, a more important point is to note that Prolog inspects facts in a program in a top-down manner. In relational database terms, this implies an ordering in the tuples of a given relation. This contradicts the definition of a relation. To avoid the problem, Educe adopts the semantic of *assert* when inserting tuples in a relation. Equivalences for *asserta* and *assertz* are purposely excluded from Educe. However, this is not sufficient in the case of general rules. To keep close to Prolog semantic, users are asked to specify an order of evaluation for the rules kept by the EDB. For example, to add the definition of *anc* to the EDB component of Educe, one should proceed as follows:

```
?- nrule( 1,
   'anc(X,Y) :- parent(X,Y) .') .
?- nrule( 2,
   'anc(X,Y) :- parent(X,Z), anc(Z,Y) .') .
```

Once this is done, the EDB component of Educe becomes transparent to those users accessing the derived relation *anc.*

Finally, a point that has some bearing on efficiency and integrity: the evaluation algorithm described above retrieves rules from *rulerel* not only when the rule is activated for the first time, but also when backtracking takes place. From an efficiency point of view this is a serious drawback. Also from the point of view of integrity, backtracking can cause some problems. In particular, the answers to a query would not be correct if another user were allowed to update part of the necessary definitions while they were still being used. Educe solves these two problems by pre-processing the top level query. Thus, given a goal to evaluate, Educe builds the whole evaluation tree for this query. Rules are then retrieved from the *rulerel* and the necessary *retr's* rules are also asserted. Only when all this information has been obtained from the EDB, Educe proceeds to evaluate the query. Effectively, the EDB is only consulted once for the necessary rules, and all the definitions needed are frozen during the evaluation of the goal. Notice that with this scheme other users are not prevented from updating the non-factual knowledge. For the factual knowledge (base relations) the EDB uses normal database techniques for concurrent access to relations.

Now we can go back to our example *ancestor* and see how recursive (and multiple) queries in loose form are handled in Educe. First, let us examine the program $*slowretr* below. This programme is a preamble to a simple but not very efficient implementation of a query evaluator for the loose DML. However, this program is capable of handling multiple and recursive queries.

```
/* $slowretr -
   it uses same syntax as retrieve
   in Atts and Boolean
*/
$slowretr( Reis, Atts, Boolean) :-
      :
   /* Reis list is obtained
      from Atts and Boolean */
      :
   $q_and_s( IntRes, Rels,
             Atts, Boolean) ,
   !,        /* never backtrack */
   $quickretr( IntRes, Atts) .
```

In this program, once the list of base relations *Rels* has been extracted from *Atts* and *Boolean* the call to $*q*_and_*s* prepares a query in loose form and executes it, saving the result in the intermediate relation *IntRes*. We do not want to backtrack past this point, hence the cut (!). It is now up to $*quickretr* to produce the answer(s), one tuple at a time. This is done by $*quickretr* by querying the intermediate relation *IntRes* using the close DML. Finally, $*quickretr* matches the values in the returned tuple (close DML) to the non-positional projection specified by *Atts*.

The first and obvious problem in this strategy of evaluation is one of efficiency. In the case of recursive definitions, each time we backtrack on the non-recursive part of a definition a new intermediate relation will be generated. This is easily solved though, by labelling the queries already answered with the name of the intermediate relation generated for it. Thus before proceeding to generate a new intermediate relation we check whether the intermediate relation has been generated for the (intermediate) query. The program for this version of *retrieve* is given below. To stress the fact that this program also handles multiple relations and recursive definitions, we call it *mretrieve*.

```
/* mretrieve -
   handles multiple relations and
   recursive definitions in the
   loose DML.
   It uses same syntax as retrieve */
mretrieve( Atts, Boolean) :-
    $evaluated( Res, Atts, Boolean)
       -> $quickretr( Res, Atts)
       ;  $slowretr( Res, Atts, Boolean) .
```

As can easily be imagined there are occasions when the above strategy to evaluate recursive queries can produce very slow responses.

5.5 Efficiency

As was pointed out in [3], users expect in the context of a Prolog interpreter to obtain a reply quickly. This reply normally corresponds to just one tuple in a relation (base or derived). During backtracking the same still holds true. By contrast, in a relational DBMS users expect a whole relation, i.e. a set of tuples, to be generated as an answer. This dichotomy between the two types of system leads to two different types of evaluation strategy for queries.

In loose coupling, queries are handed to the EDB component for evaluation. This is in effect an evaluation of queries by a DBMS. In all cases of queries involving several relations and/or recursion, and in many cases of queries involving a single relation, e.g. aggregation, a number of intermediate relations are generated during the evaluation of these queries. It is the creation and manipulation of these intermediate relations that is the cause of major overheads in the evaluation of queries in loose form, particularly in the recursive case. Even if large buffers in main memory were used for these intermediate relations, it would still be necessary to use secondary memory (slow) to store considerable parts of these relations. Also there is an overhead attached to the creation and maintenance of the schemas for these relations.

The *one-tuple-at-a-time* strategy of Prolog does not need to create intermediate relations. All intermediate results are kept at all times in main memory. This is only possible because of the relatively small size of the intermediate results required by the *one-tuple-at-a-time* strategy. In other words, retrieval of data from secondary memory only occurs when base relations are consulted.

However, as was discussed in [3], there are good reasons for using a loose DML to express queries. Also, there are situations where the *sets retrieval* strategy of DBMSs outperforms the *one-tuple-at-a-time* strategy of Prolog. In Educe, all of these reasons are considered to be important and hence queries in loose form are supported. Moreover, a number of optimisation techniques are applied to queries in loose form, in order to improve performance. In addition to the optimisation techniques of the DBMS, Educe uses its own techniques. Particular attention is given to the recursive case, since this is an area outside the scope of conventional DBMSs. Four significant cases are here discussed.

The first case arises in queries involving one base relation and the boolean condition *true*. For example, consider the relation *employee* with attributes *name, address and dept*, and the goal

```
?- retrieve( [employee.name = Name,
        employee.dept = Dept] ,
   true) .
```

This query is transformed into the equivalent "query" in the close DML:

```
employee( Name, Dept) :-
    retr( employee( Name, Dept, _) ) .

?- employee( Name, Dept) .
```

This example is generalised to the case of any base relation being queried with the boolean condition set to *true*. The built-in predicate *$whole_base_r* makes the appropriate tests to decide on the applicability of this transformation rule. Although the case described seems trivial and unlikely to be presented to Educe by users, it often arises as an intermediate step in the evaluation of a recursive definition.

The second case for efficiency improvements occurs again very frequently in recursive queries. This is the case of intermediate queries that have already been evaluated. Although this situation often arises during the evaluation of recursive queries, it is not exclusive to them.

The third case of importance occurs in conjunctive queries on a single relation. Again, this is a very common situation during the evaluation of recursive queries (top level). In this case the conjunctive query is first transformed into a normal form and then, from this normalised form, an equivalent query in close form is generated and evaluated.

The last but certainly not the least major optimisation step takes place during the transformation of loose form into close form and during the instantiation of variables at the top level. For both of these processes it is necessary to access the schema of the relation involved. These accesses are speeded up by maintaining the database schema in buffers in main memory. Obviously, some synchronisation with the copies in secondary memory is necessary. This is a common solution in a conventional DBMS and Educe has adopted it. More seriously though, whenever a tuple is retrieved from a base relation, a number of variables have to be instantiated. To do this, the list of values in the retrieved tuple has to be matched with a list of variables (typically, the variables in the projection part of a *retrieve*). The list of variables is normally shorter than the list of values and their sequences do not match. For instance, a typical tuple in our relation *employee* might be *[john, munich, toys]* and the goal might be *retrieve([employee dept = Dept, employee.name = Name]...)*. Obviously, the order and the length of the lists *[Dept, Name]* and *[john, munich, toys]* do not match. In general, to match the two lists every time a new tuple is retrieved is unnecessary. The problem is avoided by, firstly, creating a bogus list of variables, say *[X1, X2, X3]*, then matching this list only once to the projection list in the *retrieve*, and finally, each time a tuple is retrieved from secondary memory, this bogus list is used to instantiate the real variables. A lot of unnecessary sorting is eliminated at a stroke!

The optimisation steps described above have in fact led to a new strategy, the *Educe method*. This method integrates all the above optimisation steps into a

harmonised strategy of evaluation for queries expressed in loose form. The improvements obtained by the application of this method go well beyond the recursive case. In fact, the *Educe method* always give a performance close to the best of the other two methods (DBMS and Prolog).

6 Some problems

As already mentioned earlier on, the use of control predicates, such as "cut" in Prolog, causes serious problems to the optimisation of queries. Because of this we provided users with a loose DML so that interactions with the EDB appear boxed into logical units. In this scheme, optimisation by query transformation is left to the underlying DBMS. However, because of the preferred usage of the close DML by Prolog programmers, the question of goal re-ordering in the close DML had to be reviewed[21]. Ideally, we would have liked to be able to transform a goal such as

```
?- employee(Name, Dept) , ...,
        otherpred(.., Dept, ..) ,
        ..., Dept = production.
```

into the "*equivalent*" form

```
?- Dept = production, employee(Name, Dept) ,
      otherpred(.., Dept, ..) , ....
```

so that the early instantiation of *Dept* could be used on an indexed search of those *employees* in the *production* department. This sort of transformation would allow us to make full use of the data structures and the access methods used by Educe. Unfortunately, because of the likely occurrence of control predicates in *otherpred*, the suggested transformation is not possible. However, we thought that it was still possible to make an effective use of Educe's access methods (AM). The obvious case is the one where the arguments are grounded on an indexed attribute. For example, to evaluate

```
?- employee(Name, production) .
```

Educe can make use of an index on the *Dept* attribute. In order to take full advantage of Educe's AM, we stretched the previous idea to deal with range queries in the close DML. For this, we extended the Prolog syntax to accommodate these cases. Thus, expressions like

```
?- earns(Name, Salary) , Salary > 1000.
```

should be written as

```
?- earns(Name, Salary > 1000) .
```

in order to take advantage of the performance benefits of an indexed search. However, the former expression is still valid, although expensive to evaluate.

Another apparently trivial problem is the correspondence of data types in Prolog and the underlying DBMS. In Educe we chose to map

DBMS	Prolog
****	******
character string	atom
integer	integer
real	real

Unfortunately, regardless of the particular components (DBMS and Prolog interpreter) the match is not always perfect. A common problem is that while the Prolog interpreter/compiler uses tags in the internal representation of values and their data types, this same information is normally kept by the schema maintenance mechanism of the DBMS. The consequence of this is that although the DBMS and the Prolog interpreter might use one word (in the machine) to represent integers, because of tags in the Prolog interpreter fewer bits are available to store values. The same holds true for *string-atom* and *real-real*, although with slight variations. This might not seem a serious problem for completely new applications, but it is a rather serious problem when a new application attempts to use an existing database.

The question of data types leads into one more problem. Since we decided to store the intensional part as well as the extension of relations in the EDB, we needed a way of storing rules in the EDB. This sort of structured data type is not normally supported by a relational DBMS. In Educe, we solved the problem by creating a relation (*rulerel*) to store rules. The rules are stored as character strings and two new built-in predicates are used to map the rules into character strings (*swrite*) and vice versa (*sread*). Rules stored in the EDB are automatically used if no matching definition is found in main memory during evaluation of a goal. Extensions of relations kept in the EDB can be made transparent to Prolog users by defining a rule in *rulerel*.

The use of a relation to store rules in the EDB brings to the fore the incompatibility of the procedural evaluation of goals in Prolog with the concept of a relation as an unordered set. While the order of unit clauses would only alter the order in which answers are produced, the order in which general program clauses are chosen for evaluation might produce completely different answers, if any at all. For instance, choosing the recursive rule first for the evaluation of the classical *ancestor* definition, could lead to an infinite loop, while choosing the non-recursive rule first would produce the correct answer(s). To get around this problem, users of Educe have to specify the order in which rules stored in the EDB should be used during evaluation of queries.

Still on the subject of rules in the EDB, two additional problems. As already mentioned, once rules are stored in the EDB they can be used like any other

rule. A simple way of achieving this is to modify the top level of the Prolog interpreter, so that whenever no matching definition for a goal G can be found in main memory, the following programme is executed:

```
interprete(G) :-        /* evaluate G */
        functor(G, Label, N) ,
        /* find rule in EDB */
        rulerel( Label, N, _, RuleinChars),
        /* convert string into rule */
        sread( RuleinChars, Rule) ,
        (Head :- Body) = Rule,
        G = Head,
        call(Body) .
```

Imagine that, among the several clauses defining G, at least one is recursive. The first problem is one of efficiency. In order to evaluate G, it is necessary to retrieve the recursive clause several times. The second problem is even more serious in a multi-user environment. Consider the case where one user is evaluating G, while a second user is modifying the recursive clause. This leads to a situation where, during the evaluation of G, two different recursive clauses are used. This evaluation of G might thus produce seriously wrong results. In order to solve these two problems, we decided to pre-load all the necessary definitions into main memory before starting the evaluation of the top goal. On return to the top level interpreter, and when no more answers are required-or no more answers can be found, the pre-loaded definitions can be erased from main memory. This scheme effectively freezes definitions until the evaluation of the goal(s) at the top level is completed. In addition, it solves the efficiency problem, since now only one access to each required clause in the EDB is made.

However, the above scheme is not enough. It would fail to evaluate correctly queries like

```
?- nrule( 1, 'p(X) :- q(X) . ') , p(Z) .
```

even if the internal/external DB contained

```
q(a) .
q(b) .
 :
 :
```

The reason for the failure is due to the attempt to pre-load the definition for p, before the new rule (*nrule(..)*) is added to the EDB. Our eventual solution was to incrementally load the definitions from the EDB as they are required. This solves both of the above problems, efficiency and concurrent access, and at the same time it maintains the order of evaluation prescribed by Prolog.

7 Conclusions

Although a number of very significant problems have been solved with the implementation of Educe, we can still see some fundamental shortcomings. These we believe are inherent in any system constructed by the coupling/integration of Prolog and a DBMS.

A primary goal in the implementation of Educe was to provide users with a logical programming system for the construction of large KBMSs[21]. We want to keep the external appearance of this system as close to Prolog as possible. This was dictated by the early development of other prototype systems within our research group. These prototypes were all originally implemented in Prolog[5]. Thus, in the particular case of the close DML we wanted Educe to be transparent to Prolog users. Let us start by examining the extent to which this proved possible.

The first case we examine is one of semantics. The Prolog user can use *asserta* to add a new fact or rule at the beginning of the clauses for a predicate, *assertz* to add them at the end, and 'assert' if it doesn't matter. For example if one writes in Prolog

```
?- asserta(r(1)) .
?- asserta(r(1)) .
?- assertz(r(2)) .
?- assertz(r(3)) .
?- assert(r(a)) .
?- assert(r(b)) .

?- r(X) .
```

the answers 1, 2, 3 will be returned in order, but one does not know when the answers a and b will be returned, either before, after or amongst the other answers. In Educe, if a user wants to impose a specific ordering on any set of facts in the EDB they must be held as rules. This is because relations by definition are an unordered set.

The second problem is caused by the differences and limitations in data types supported by the component sub-systems (Prolog and EDB). Even if there is a unary relation *r* in the Educe database, the user cannot insert into it the fact *?- r(head(a))* because *head(a)* is a compound term. There is no database type *structure* in which to store Prolog structures. The reason for this is that the types provided by Educe are precisely the Ingres types. When an Educe relation is created, its definition is passed directly to Ingres. This avoids the more serious problem of having to maintain duplicate schemas, one in Prolog and one in the EDB. One possible solution that we have considered is to have intermediate Educe types which can map onto the underlying Ingres types. This can be implemented by adding a new system relation for this purpose to the EDB.

The third is the insertion and deletion problem. When a Prolog clause is

asserted it can be either at the beginning of the clauses for a particular predicate (*asserta*), at the end (*assertz*), or its position may be unspecified (*assert*). To insert a rule into Educe it is necessary to give it a number (distinct from all the other rules for the same predicate). To retract a rule the user must either supply this number, or take the textual representation of the rule, append a full stop, convert it to an atom and then do the deletion from the database. Clearly insertion/deletion of facts and rules into the Educe database cannot be transparent since Educe must still support the standard semantics of assert and retract. However the only reason for requiring the user to assign a number to each rule is because it is used to force an order in the otherwise unordered relation *rulerel*. This is one more example of the contradiction between the procedural nature of Prolog and the unordered set concept of relational DBMSs.

Unlike the three previous ones, the last problem is not about transparency. It does not affect ordinary Prolog calls. However, we feel it is a related issue and hence it has been included here. The extension to Prolog in the close DML allows a call of

```
?- retr(r(A,B)), B>20 .
```

for example, to be expressed as

```
?- retr(r(A,B>20)) .
```

In general any comparison can be expressed in the embedded form. In order to gain the efficiency advantage that Educe provides for such queries, the user needs to use the extended syntax. Unfortunately, the syntax used by Educe in the extension is ambiguous. For example, to store two clauses for the predicate r, one in the relation *r*, and one in the rule relation (*rulerel*), we might add the fact *r(fred,20)* in the relation *r*, and the rule *r(X,50) :- manager(X)*, in the rule relation. To make the close syntax transparent we can add an extra rule *r(X,Y) :- retr(r(X,Y))*. Now the query

```
?- r(X,Y) .
```

will succeed, returning {X = fred, Y = 20} and {X = M, Y = 50} for every manager M. Suppose the user wishes to take advantage of the extended syntax. He can ask

```
?- r(X, Y>10) .
```

and this will call the subgoal

```
?- retr(r(X,Y>10)) .
```

which will efficiently return the answer {X = fred, Y = 20}. It will not, however, return {X = M, Y = 50} for any manager M because the other rule for r has head *r(X,50)*, which will not unify with *r(X,Y>10)*.

Thus the user has to express the rule as:

```
r(X,Y) :- var(Y) , !, manager(X) , Y = 50.
r(X,Y) :- integer(Y) , !, Y = 50, manager(X) .
r(X,Comp) :-    comparison(Comp,Y,Op,V) ,
                Y = 50, call(Comp) ,
                manager(X) .
comparison(X>Y,X,'>',Y) .
comparison(X>=Y,X,'>=',Y) .
...
```

to resolve this particular problem.

Two more problems worth mentioning are debugging and error handling. Usual Prolog facilities for debugging are just not satisfactory in the context of large knowledge bases. Just imagine trying to detect a loop involving a few thousand tuples in a relation which is used in a recursive definition of a predicate. Spying and tracing are too primitive for such a problem. The error handling problem is due to the different treatment and recovery techniques that the two components (Prolog and EDB) use. In Educe we tried to be consistent by adopting the Prolog treatment. Unfortunately, this is not always possible, particularly in the case of the coupling, where there is little control over what is generated by the internal modules of the EDB.

From the discussion on transparency, it is clear that although Educe is a close approximation to Prolog it is not an extended Prolog system which can store facts and goals on disc. However, it is indeed an efficient logic programming system for constructing large KBMSs, as [20,21,5] demonstrate.

In this paper, we have undertaken a critical revision of our work on Educe, a logic programming language for the development of KBMSs. The version of the system here described has been successfully tested with the implementation of KB-2[21]. We have also tested the performance of Educe with queries on reasonable large relations and several levels of recursion[4]. Response time for the first answer and for backtracking answers is never more than a few seconds. We do not expect any significant deterioration in performance on very large knowledge bases.

Based on our past experience we have decided that the next version of Educe will move towards logic programming, and not towards Prolog. This is a departure from our previous adherence to Prolog. The new system, Educe* is modular, uses new evaluation strategies and aims for completeness in the recursive case[20]. Also, in this new system we are relying much more upon our own access methods, so that we are less dependent on the underlying DBMS. All of this, plus the addition of structured data types, should resolve the main incompatibilities between the EDB and the Prolog component. Indeed, Educe* is no longer a marriage of these components, but a self standing system.

Acknowledegment

I would like to express my gratitude to all the members of the Knowledge Base Group at ECRC, for their many helpful comments and discussions.

References

1 NICOLAS, J.-M. and YASDANIAN, K.: 'An Outline of BDGEN: A Deductive DBMS'. Information Processing 83. Elsevier Science Publishers B.V. (North-Holland), 1983, 711–717.
2 BOAS, H., BOAS, P. and DOEDENS, C.: 'Extending a Relational Database with Logic Programming Facilities'. Technical Report, IBM INS- Development Center – The Netherlands, 1984.
3 BOCCA, J.B.: 'EDUCE – A Marriage of Convenience: Prolog and a Relational DBMS'. Internal Report KB-9, European Computer-Industry Research Centre, Munich, September, 1985.
4 BOCCA, J.B.: 'On the Evaluation Strategy of EDUCE'. In: Zaniolo, C. (editor), Proc. 1986 ACM-SIGMOD International Conf. on Management of Data. ACM, Washington, D.C., USA, May, 1986.
5 BOCCA, J.B., DECKER, H., NICOLAS, J.-M., VIEILLE, L. and WALLACE, M.: 'Some steps towards a DBMS based KBMS'. In: Kugler, H.-J. (editor), Proc. 10th World Computer Congress. IFIP, Dublin, Ireland, September, 1986.
6 CHANG, C.L. and WALKER, A.: 'PROSQL: A PROLOG Programming Interface with SQL/DS'. In: Proc. of the First Int. Workshop on Expert Database Systems. Kiawah Island, South Carolina, USA, October, 1984.
7 BOCCA, J.B.: 'EDUCE – User Manual (Internal KB Report)'. ECRC, Arabellastr. 17, Munich, W. Germany, 1986.
8 DUCASSE, M., FAGET, J. and GUMBACH, A.: 'FROG – Implementation of a Language merging Functional Relational Programming Styles, via an Object Type Driven Evaluation'. Laboratories Marcoussis, 1985.
9 GALLAIRE, H.: 'Logic Programming: Further Developments'. In: Proc. 1985 Symposium on Logic Programming. Boston, USA, July, 1985, 88–96.
10 KUNIFUYI, S. and YOKOTA, H.: 'PROLOG and Relational Data Bases for Fifth Generation Computer Systems'. CERT Workshop, France, 1982.
11 LI, D.: 'A PROLOG Database System'. Research Studies Press Ltd., 1984.
12 UNIX's notes (file). Announcement of 'near' availability, 1985.
13 NAISH, L.: 'MU-PROLOG 3.0 reference manual'. Melbourne University, Computer Sc., Melbourne, Australia, 1983.
14 RAMAMOHANARAO, K. and SACKS-DAVIS, R.: 'Recursive Linear Hashing'. ACM Transactions on Database Systems 3(9), September, 1984.
15 STONEBRAKER, M., WONG, E., KREPS, P. and HELD, G.: 'The Design And Implementation Of Ingres'. ACM Transactions on Database Systems 1(3), September, 1976, 189–222.
16 STONEBRAKER, M.: 'Inference in Data Base Systems Using Lazy Triggers'. In: Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, Islamorada, Florida, USA, February, 1985, 295–310.
17 Unix Programmer's Manual. 4.2 Berkeley Software Distribution (copyright 1979, Bell Telephone Laboratories, Inc.) edition, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, USA, 1983.
18 VASSILIOU, Y., CLIFFORD, J. and JARKE, M.: 'Access to Specific Declarative Knowledge by Expert Systems: The Impact of Logic Programming'. Decision Support Systems 1(1), 1984.
19 VENKEN, R.: 'The Interaction between Prolog and Relational Databases'. Unpublished, Early, 1985. Report on ESPRIT Pilot Project 107
20 VIEILLE, L.: 'Recursive Axioms in Deductive Databases: The Query-subquery approach'. In: Proc. First International Conference on Expert Database Systems, Charleston, South Carolina, USA, April, 1986.

21 WALLACE, M.G.: 'Reconciling Flexibility and Efficiency In A Knowledge Base Implementation'. Internal Report KB-8, European Computer-Industry Research Centre, Munich, September, 1985.
22 WARREN, D.H.D.: 'Logic Programming and Knowledge Bases'. In: Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, Islamorada, Florida, USA, February, 1985, 69–72.

The semantic aspects of MMI

"So advanced, it's got to be simple."
Advertisement for Canon T70 camera

John M. Pratt
European Computer-Industry Research Centre GmbH, Munich, West Germany

Abstract

The factors improving the comprehension by a user of a domain of knowledge, when assisted by a computer based decision support system, are analysed. The classification of the domain into a model, the presentation of the model to the user, and the relevance of Fifth Generation techniques are discussed.

1 The context

1.1 Introduction

ECRC has been established to investigate *Computer-assisted Decision Making* techniques. In the most complex cases such CADM needs powerful reasoning techniques, and therefore is a challenging application of the most advanced computer systems. Topics such as knowledge representation and manipulation, logic programming, and inference crunching are common factors to all the work at ECRC, and have a lot in common with the "5G" programme.

The MMI dimension of the work is that the users, *who are the decision makers*, may not have the ability to access or comprehend such knowledge without assistance. It is important to recognise that humans should make decisions, not machines. In spite of all the claims about the potential power of computer-based systems, background knowledge of educated users is still essential to achieve optimal decisions. The MMI group aims therefore to investigate methods of providing and promoting such assistance, and to measure the effectiveness of CADM in terms of the improvement in decisions made by the users.

Thus, users are expected to be motivated to understand and manipulate some domain of knowledge relevant to their current situations. Individuals do not wish to be aware of the workings of the tools they are using, except when the tools are being used beyond their designed limits. For example, persons driving a car are normally concerned with their situations on the road and not by the operation of the car. But if they ask too much of their car

they expect to be made aware of the danger by an exception report in sound, vibration, reaction of the steering wheel, or warning lights.

Therefore, it is argued that MMI is concerned with three aspects:

- **The domain**, which contains the problem which the user is attempting to understand.
- **The model**, which is the representation of that domain in a machine.
- **The presentation**, which is the image of that model, and is the means whereby the user is communicating his wishes.

The prime measure of the success of MMI is that of the degree of **transparency** with which users comprehend the operation of the domain. That transparency is provided by the model and the presentation.

1.2 The domain

This is defined as the body of knowledge which users wish to comprehend. In some professions, accounting for example, the domain is built around precisely specified and systematic concepts. In many other domains this is less evident, and the difficulty users face is grasping the real nature of the imprecise nature of the domain. Clarification of the concepts lies firmly in the domain of *human* researchers and tutors, who ought to extract the essence of the domain for other users. Of course, it is not always possible for that extraction to be uniquely prepared and the domain then exists as a volume of prior examples from which users are expected to extract their own interpretation. The knowledge about the domain will be recorded in some language, which is normally capable of ambiguity and contradiction.

It is evident that the ambiguity of natural language is a valuable feature of some domains. The art of politics relies heavily on multiple interpretations, and the legal profession is built on the idea of new interpretation of old law. Human expressions, such as "thin", "very thin", "next to", "east of", are all logically imprecise, and yet can convey significant meaning in appropriate contexts.

However, there is increasing recognition of the power of *Formal Language* to capture such knowledge, and in some cases, knowledge about a domain can be expressed only in formal terms. For example, knowledge about dynamics of motion can best be properly expressed with calculus, and thus that particular branch of mathematics should properly be classed as the *language* of that domain. It should be noted in this case that the knowledge was not created *until* that language was available.

A further example is the use today of languages based on set theory to describe the world of "office automation". In this case, users need to be able to precisely express the movement of information in an office and need a language which provides the concept of membership of a group. Thus the idea of a domain is closely associated with that of a particular language.

Ideally we should include many branches of mathematics as languages, and encourage users to use them when relevant.

Where a group of users have a need to share knowledge then it is important that they use a common language. That language evolves within their domain, and records the extent of the knowledge of the domain. In his book titled *Truth and Method* the philosopher Gadamer (1975) proposed that language and the understanding of a domain are mutually dependent in that individuals, in understanding their world, are continually involved in acts of interpretation based on previous understanding, which includes assumptions implicit in the language that individuals use. That language, in turn, is learned through activities of interpretation. Individuals are changed through use of language and the language changes through its use by individuals.

1.3 The model

A model is conventionally defined as the representation of a domain, defined in the language of that domain. As such, it is an approximation of the domain, and has limits to the represented knowledge. Ideally, users comprehend the domain through the model without being aware of the implementation of the model. The model may have been created by system designers, rather than end-users, and the former should be aware of the approximations they have made, and should try to anticipate the alternate sets of approximations that users might wish to apply.

The model will evolve as the users interact and refine their understanding. Such evolution needs to be managed such that old instances of the model are not prematurely discarded. For example, a model of a document, i.e. the files of text, will evolve through many issues. The discarding of old issues should be under users' control. Therefore, from users' perspectives, models should be *manageable and flexible.*

In order to have a focus for the research we have attempted to clarify the term "Complexity". In human terms we propose that such complexity can result from:

- Structural characteristics, as represented in the schema of an Entity-Relation database.
- Quantity of entities, such as a large set of geographic data.
- Value interdependence, where the relative importance of the values and their interaction are significant.
- Question characteristics, where the problem to be solved is not easy to comprehend.

A characteristic of the model is that of multiple dimensions, which mutually interact. Users would obviously prefer a simple one or two dimension view of the domain, which can be simply expressed as a table of numbers or a Cartesian graph, However we have assumed that in the applications of

interest such simplicity of extraction is not possible, and that three or more dimensions of comprehension are desirable. Our challenge is to find ways of helping the user find such comprehension.

1.4 The presentation

This is the interface with the users. Many different communication methods can be used, and the success of a particular method is measured in terms of the accuracy and efficiency with which users can manage and manipulate models, and thereby comprehend the domain.

This interface is built to take advantage of the abilities of the users, which are both physical and mental. Direct manipulation of objects on a visual display screen, in the manner of Macintosh, creates a communication channel which is limited to visible objects. In contrast, the expression of commands fitting an abstract model, in the manner of UNIX, makes maximum use of the mental abilities of users, but, therefore, has greater scope. In both cases, the interface should be maximally *transparent* and ideally users should not be aware that they are operating the interface.

1.5 Reversibility

In some domains reversing of action may be impossible, because potential for catastrophic changes exists in the domain. In such circumstances models ought to provide reaction to users' actions which indicates that they are "driving dangerously close to the limit". This implies that the model of the domain needs to be testable without commitment. In this respect, the model is more manageable than the real world.

1.6 Some examples

To illustrate these aspects, let us review some examples.

1.6.1 A spread-sheet calculator application: Here the domain is typically that of a business world. The user is concerned about quantities of resources and money. The complexity of the domain results from the manner in which those *values* interact.

The language of expression is mathematically quite simple, with algebraic expressions sufficing to represent the relations and quantities. The user is also aware of time as a dimension in the domain, with each year's results progressing from the previous year.

Algebraic expressions, or equations such as:

Gross Margin − Production Costs = Profit
Labourcost[this year] * Inflation = Labourcost[next year]

can be used directly to build a model.

The grid of cells on the visual display form the presentation, and for convenience the names of the resources are often assigned to lie on rows and time is represented by the columns. Generation of other types of presentation, such as graphs of sets of the values, is normally available. The rows and columns also provide obvious grouping of values, which can then be the subject of further expressions, such as "average value of a group".

Thus the domain, the model, and the presentation all fit smoothly together, and users are able to directly appreciate what their domain is doing. It is not surprising that such facilities are popular, given such directness.

However, as their horizons expand, users will become aware of the problems of managing bigger models, where the quantity of values and expressions increases to the point when they cannot be easily managed. Integration then becomes a limitation.

Observation of users has shown that, even with such simple models, users understand the domain better if the operation of the model is "explained". For example, the machine can compute the values which satisfy the equations in any direction. Given some values, it can therefore automatically deduce the remaining values, and the most significant flows of value, and highlight them for users. In such cases users report a significant improvement in the transparency of the machine and their understanding of the domain.

1.6.2 A structured database: Here we have a domain which is the world of objects and their relationships. The complexity of the domain is that it is very difficult to classify objects and relations in a way which is equally applicable to all uses. A small example of the difficulty is "What is *one* object", for there is no precise point at which reality can be divided into discrete objects. An object at one scale is really a set of objects at another scale. That is reality.

An example of such an application would be the representation of the social and public activities in a locality. Figure 1 below shows a pictorial view of a section of such information. Here the language of the domain is that of the general public, but note that the comprehension of the meaning of simple words such as "stage" is highly subjective. Thus the classification of items into the structure shown is not unique, and other classifications would have been equally valid. The figure also shows that the model has a structure, and emphasises the relations which have been recorded between objects.

A presentation of that model could be abstract, tabular, or the graphical diagram shown. In the first case users would need to have learnt a particular query language, which would possibly extend to include symbolic manipulation and logical expressions. They would need to have a mental image of the model to form sensible queries, such as "Find events at place X, and retrieve list of nearby restaurants".

In the second case users directly operate on the items in a predigested list,

which does not give them a view of the model or the reality behind it. This would often be the case when the users consulted printed literature, where the information has already been grouped by the publisher. If it is presumed that the relations between objects are relevant to a user's problem, then such a presentation is obviously less than helpful.

In the third case users interact with a view of the structure of the model as shown in the figure. This gives a more transparent view of the model, and

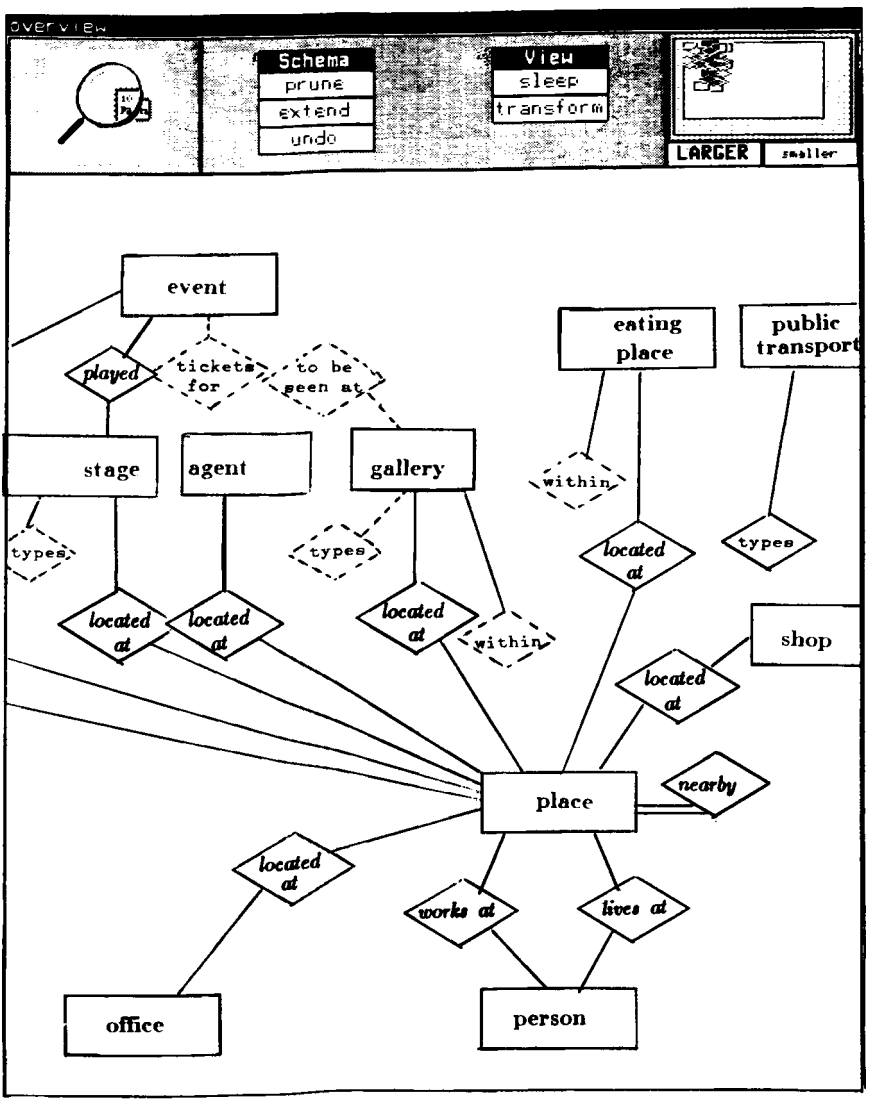


Fig. 1 A graphical view of a model

enables users to directly manipulate the objects and the relations between the objects and queries can be expressed by the selection of combinations of objects and relations. In addition it should be noted that the visibility of the overall arrangement should provide insight into the way reality has been classified.

1.7 Summary of the context

Thus MMI is concerned with helping human users to comprehend a particular domain. In such a context machines, and in particular computers, act as communication agents. These should ideally be as transparent as possible, and where their design is limited users should be able to manage their characteristics to expand their limits. The language of communication should be chosen to best express the character of the domain.

Since the domain knowledge has previously been gathered by a human act, in the limit the interaction is limited to that knowledge. "MMI" could then be interpreted as "Man to Man Interaction".

2 Characteristics of the components

A total system can be divided into three sections, namely the human user, the model of the domain, and the presentation (which is the communication medium between the first two sections). The essential semantic aspects of these are as follows.

2.1 The human

The objective of MMI research in general is to find the methods which allow users to achieve the greatest accuracy in the shortest time, and with the most satisfaction and deepest comprehension. In the particular case of decision support we are concerned with situations where the cognitive demands of "thinking about the problem" dominate the more automatic activities such as reading values from a graph. We are therefore concerned with finding consistent measures of cognitive processes, both objectively in terms of the accuracy of decisions, and subjectively in terms of the feelings of users. The following characteristics of users have been found to be important to the decision making process.

2.1.1 Perception of the whole: Humans have the ability to best perceive detail by using its context in a whole picture. For instance, the perception of the colour of an object is known to be determined by reference to the colour of all the surroundings, and thus not to be dominated by the colour of the incident light. This is known as "colour constancy", and implies that the brain does not process individual items in isolation. One also could note that the comprehension of the meaning of a name is helped by noting its relation to neighbouring names, and forming a view of the boundary between them. This ability is the foundation of the success of whole screen interaction.

2.1.2 Abstraction: Humans operate primarily by *recognition of patterns and structures.* Since structures within a domain may be naturally multi-dimensioned, it is not always possible to make an appropriate visual image of structures. In some cases users can operate better with a memorised image of the model they are manipulating. An example could be that of the players of "adventure" games who are navigating in a world more complex than a visual image could convey. Alternatively, consider that skilled users of UNIX seem to prefer an abstract environment, which offers more flexibility than a direct visual representation.

It also seems to be common for problem solvers to think better about a problem whilst not actively looking at anything, and this could suggest that the act of observing inhibits analytical thought.

2.1.3 Satisficing: Previous work by H.A. Simon proposed that users will, when presented with a complex problem which is beyond their power of analysis, make a decision which they *feel is likely to be satisfactory,* without being able to rationally argue a justification. The term "satisficing" attempts to capture this human quality. Such decisions, made by rules of thumb or intuition, are one of the strengths of humans. Whilst this ability is potentially very valuable, and could even be taken as the reason for allowing humans to make decisions, the decisions thus made can sometimes be non-optimal or lead to catastrophic situations. Users should therefore be encouraged to explore models to find surprising interactions.

2.1.4 Satisfaction: Users have subjective feelings about their work, and "satisfaction" is a quality which can be measured by a questionnaire. Users appear to gain more satisfaction from making the "right decision" than from the style of the interface. Experiments have shown that subjects report greater satisfaction when the operation of the model is explained, even though they do not at the same time achieve greater accuracy of decision.

The quality of documentation of the system is also an important contributing factor to the user's satisfaction.

2.1.5 Information overload: We are expecting users to be capable of operating in a challenging environment, but one of the biggest dangers they face is that of information overload. In such situations, human decision making can sometimes degenerate. (They can't see the wood for the trees.) We have found that when users are presented with an optimum amount of information they report that (1) they feel the problem space to be more visible, (2) they feel more satisfied with their work, and (3) they attribute their performance to themselves rather than the machine. If the amount of information presented is either *more* or *less* than this optimum all these measures degenerate.

We must therefore include in our system design some control on the quantity of information being made visible in order to prevent non-optimum situa-

tions arising. A current example could be that a large screen, multi-window display can hinder some types of structured tasks which do not need multiple sets of information simultaneously.

2.1.6 Training and experience: The level of users' experience will obviously affect their use of the system. They may have prior experience of the domain and be skilled in the particular language of that domain. In such a case, it would be frustrating for them to communicate in any other manner. On the other hand, if the domain is novel to users, a vital aspect of the system ought to be the training of users to learn the language of the domain. However, in neither case should the internal characteristics of the machine dominate the communication, and the domain language should be as visible as possible. That ideal may not be reachable without some training of users in the techniques of communicating via a machine. The exact mechanism of communication, be it keyboard, mouse or speech, is not relevant to this discussion, but a willingness to learn a reasonable level of skill is essential.

The new power of analysis that the machine provides introduces the need to properly express and control the analysis. For instance, the concept of taking a group of objects and forming the average of their individual values implies several manipulative and expressive skills. Such skills could be exercised by direct manipulation of visible objects, or by a symbolic expression referring to the objects and functions by name. The latter technique offers more flexibility, but is a linguistic skill which needs to be learnt, and it is unfortunate that such skill is normally associated with the full complexities of mathematics. Whilst most people would be confident that they could use arithmetic signs, such as in:

Profit = Sales – costs

it is unpopular to propose that they should use "brackets" to express the idea of a group:

Members are (joe, fred, jim, george)

Such primitive linguistic skills should be encouraged, they are "mental tools", and the alternative is that man will remain in a world of manual tools.

2.1.7 Personality: It is possible to measure personality characteristics of users, and these sometimes correlate with the best means of interaction with a decision support system. For instance, by subjecting users to a prepared questionnaire the following characteristics can be estimated:

- The preference of the users for "learning by studying", rather than "learning by doing".
- The goal-orientation strength of the users.
- Their attitude about the use of plans. {Planfulness.}

Such estimates can then be used to optimise the style of the interaction.

Further, personality measures of the individual users are possible and could be consulted along with a record of the actual use exhibited by users, and any measures of their level of experience, to formulate a "User Model". This model can then guide the presentation in terms of style of presentation and the level of detail, and thereby optimise the communication. The degree to which this is practised is a matter of current research and debate, and it leads to the concept of a *tailored* environment, as distinct from a canonical, or average one.

2.2 Characteristics of models

Although well known to information scientists, the following concepts about models need to be understood by users, since they determine their abilities to systematically comprehend their domain.

2.2.1 Entities – the subdivision of reality: Entities (or objects) are a human concept used to define a model of reality. Different people will often use different categorisations in forming their model. Since reality is a continuum and entities are discrete, a set of entities can never be the only unique model, but is often a usable approximation. The psychologist Jaensch stressed in 1930 that there is a need for tolerance in allowing different viewpoints, and that a closed, rigid system would generate a false sense of security.

The model designers identify the particular choice of categorisation, and give names to the entities thus conceived. However, the meaning of the individual names is not always clear, and is best understood by the users by seeing the whole picture, i.e. the total set of names. They then perceive the boundaries between the categories. Further, a long description of the meaning of the name could be recorded along with the name, to help clarify the scope of the name, and this may then help the designer and the user communicate their concepts. Note that this would even be desirable in the case of a spread-sheet calculator application, where the user and the designer are probably the same person, for the use of a short-form name of an entity could result in inaccurate classification in a later re-use of the application.

2.2.2 Attributes: The identification of attributes, or properties, of an entity is an important step for users, for in taking this action multi-dimensionality is introduced to the model. It is natural for users to wish to form groups of entities with similar values of attributes, but of course the membership of such groups differs from one attribute to another. The subdivision of the total reality according to the value of a particular attribute has in effect created a further model.

There are many different ways in which a given set of things can be grouped, including by spatial, temporal, and abstract concepts, such as ownership and inheritance. In some of these cases the value of the attribute is implicit in the order of entities along that attribute, and as such is not explicitly recorded.

The concept of *precision* is also often overlooked. It is common in physical science to record not only a value but also the accuracy to which it is known. Such a concept should ideally be included in any model with numeric values. In models of logical relations one could also expect a value selected from (true, false, not-relevant, unknown) to be used.

2.2.3 Order: The concept of order is an essential part of the users' comprehension. Given any particular attribute it is often just as important to conceive of the relative positions of two entities along that dimension as it is to know the actual value. The concepts of lists, queues, and number rely on that understanding. Thus the model should record the relative positions of entities along each particular attribute dimension, either explicitly by value, or implicitly by reference to another entity.

For efficiency of expression it is also possible to record rules, such as *Every tree is made of wood.* This declaration is then understood to be true within the reality modelled, and creates an automatic inherited value for an attribute, generated by being a member of the group "trees".

2.2.4 Interaction with models: Some users can gain significant comprehension by active manipulation and exploration of the model. This is a result of their ability to learn by observing the consequences of their actions, rather than by study of a static image. In general such manipulation takes the form of users modifying existing structures or values, and then allowing the model to adjust any dependent values. A colloquial expression for this is "what if" interaction. Naturally, users are a lot more exploratory if they are confident that such interaction will not result in any loss of information and that they are able to reverse, or *undo* their action.

Users also need to be able to select a subset of a model called a *view* which then narrows the field of comprehension. A more precise selection, or a query can then be matched by the model. In the case of numeric models it is also valuable for the system to be able to search for values which allow the result of the evaluation of a model (a goal) to be satisfied.

2.2.5 Evolution: Naturally, a model will become in time an inappropriate description as the needs of the user evolve. The model must therefore be *flexible* and capable of growing to match the users' needs. One advantage of the "spread-sheet" calculator is its ability to be extended by users as the problems evolve. Its weakness is in the limited scope of expression, and the consequent inefficiency.

2.2.6 Summary of model characteristics: The points above are the concepts about models which users must understand, since they represent the manner in which they or someone else described their reality. The issues discussed are fundamental to the act of modelling a domain, and are not characteristics of the machine itself.

2.3 Characteristics of presentation

Given that users have a need to interact with a model, the presentation or interface mechanism has the goal of providing that interaction in as *transparent* a manner as possible. A good interface should not be noticed! The following issues are relevant to the choice of a presentation style.

2.3.1 Views: If we assume that the models are large and complex, then we cannot avoid the suggestion that users need to reduce their problem space by the selection of "views". These are selections out of the total space, determined by the expressed context of users. They are in general abstract selections, expressed in terms of categories of entities, or by range of value of the attributes. More than one view is permissible at a time, but *interaction with one of the views should be reflected in the others simultaneously*, if relevant.

2.3.2 Predictability: Users should be expected to change their focus of attention without warning, and consequently the interface needs to follow their focus, from object to object, and from view to view. This has important implications to the construction of the interface, since it prohibits the use of traditional "dialogue sequences". The interface should instead be thought of as a mechanism which is capable of *recognising* the meaning of the actions of the user in as flexible a sequence as possible.

2.3.3 Direct versus indirect manipulation: There appear to be two contrasting styles of manipulation. First, the directly manipulated visual interface and, second, the indirectly manipulated symbolic interface. Much has been said and written about the advantages of direct manipulation, and products like the Macintosh exhibit its virtues and vices.

The concept of direct manipulation is that the model of the domain is presented in a visual form, and that users can express their wishes by selection, movement and placement of the images. Selection normally implies pointing with a mouse, although other forms of pointing at an image could be used. The important characteristic of such pointing is the precision of the positioning, and given typical users' coordination abilities the quantity of objects that one can see and manipulate is limited.

Such an interface is also bound to be limited to those aspects of a domain that can be visualised. Although this need not merely be limited to physical objects, when a model contains multiple dimensions of grouping the task of drawing an image may be impracticable. In turn, designing the model by such an interface could limit the scope of the model for adequately representing the reality.

The transparency of such an interface is high in simple models, but decreases rapidly as the models become more difficult to visualise because of their complexity and size.

In contrast, an interface which relies on symbolic expression has a much wider scope. Symbolic pointing is unlimited in scope, and all entities and attributes can be referred to by name, and abstract actions such as "sort" can be activated. In fact, one could note that over the last two thousand years we have been evolving an increasing skill in symbolic expression, and it would be unfortunate if this skill was not utilised. It may also be relevant to note that users who are attempting to comprehend a domain need finally to have an abstract comprehension, and good examples of symbolic manipulation should be used as a stimulus to clearer thought.

Transparency is enhanced by the appropriate use of both approaches, with direct manipulation being one of the tools available to manipulate symbolic ideas. The choice of the mixture of the two styles therefore needs to be carefully balanced, taking into account the abilities and preferences of the user.

2.3.4 Speech, voice, and sound: The use of sound and speech is mainly not relevant to the semantic aspects, except that one must be aware of its limitations.

Voice input of individual words is in effect a form of manipulation, where the objects selected may, or may not, be visible. The exact sound used is not relevant to its meaning, as long as it is distinguishable. The objects selected must however exist for the selection to be meaningful, and this implies that the interface may need guidance from the model as to the context of the communication, in order to correctly decode the input.

The use of sound or speech for output is limited by the sequential nature of the communication. It requires humans to remember a group of sounds, and then form an abstract model of the meaning.

The use of "Natural Language" can only be discussed in the context of the domain. Even human to human communication can only be given meaning if the context is known by receivers. Therefore the designer of the model of the domain must identify the domain specific language which is appropriate for the domain. Any form of language which is less precise than the model implies is not likely to help users comprehend the domain.

2.3.5 Users' preferences: Users' style will vary, and there is increasing evidence that an interaction controller could make choices on presentation detail, according to measurable actions and attitudes of the user. The choice between use of a symbolic command language and a menu based selection is an example. Further choices could be made between the use of graphical charts or numeric lists, where there is evidence that users exhibit varying success of perception of value according to their personality and the style of presentation.

2.3.6 Summary of characteristics: The combination of the above semantic features of humans and machines should result in a total system of great

power, where the human abilities of pattern recognition and abstraction are amplified by the ability of machines to accurately manipulate large volumes of detail. One should view a Man-Machine System as a symbiotic combination, drawing on the strengths of both components. In contrast, to expect the machine to emulate the strengths of a human is an approach likely to fail.

Both the Man and the Machine should be expected to *evolve* as the understanding of the domain increases. Flexibility of structure and language is therefore one of the most valuable features, and yet is the most difficult to manage. The communication between man and machine needs transparency of interface, and should permit multiple concurrent views to be active.

3 The relevance of Fifth Generation techniques

3.1 System design

The concept that has been used throughout this discussion is that an "application" naturally divides into two differing specialities. Namely, model design and interaction design. The former is concerned with the use of increasingly powerful methods of knowledge representation, and the latter is concerned with the construction of real-time interactive control mechanisms.

3.2 Knowledge bases

The implementation of the domain models will be assisted by the availability of knowledge base techniques, particularly those using logic programming as the means of manipulating the structures. The concepts of entities, attributes, relations, and rules all map directly on to the emerging techniques. Naturally, all the techniques needed to maintain the consistency and efficiency of the structure are also needed, even though normally they are not visible to the users.

3.3 Interaction design

The complexity of interaction design comes from two sources. Firstly, there is the huge variety of human types. A programmer has previously had to make assumptions about the users, and only by following prior examples made the variety manageable. This has resulted in mediocre interfaces. One can easily improve this. One can cater more for the lowest common denominator, which assumes the least capability of users (e.g. the Macintosh – the ultimate User Friendly System), but at the expense of making the environment of more experienced users less efficient. Alternatively, one can construct an adaptive interface, which ideally is built to grow naturally on top of a good minimum, offering different levels of abstraction to suit all users. Such adaptive interfaces are naturally more complex in internal structure, but force the designer to distinguish between the semantic and syntactic aspects of the interface.

Secondly, computers have to allow humans to be humans, to make mistakes, do unpredictable things out of sequence, and still optimise the overall interaction. That is, machines need to adapt to humans, not the converse. This implies an interface control structure which is driven by the actions of users, and yet has some sense of the most appropriate reaction at any particular point in the application. Whereas traditionally, a state of a system is implicit in the history of the interaction, ideally the state of the system must be explicitly recognisable by the interaction control.

3.3.1 The design of control: Obviously, such stress on the importance of recognition indicates that future machines should contain recognition, or pattern matching mechanisms as primitives. Prolog language is currently being used to good effect, but it normally implies a particular search strategy (depth first – left to right), and this can be inefficient in some circumstances. Breadth first matching by coarse criteria is much more useful to interaction control, where matching is basically a matter of successive refinement of match. For instance, if a cursor is not within a window bound then all the detail within that window can be safely ignored during the search.

Optimising compilers would normally try to find a correct implementation of search sequence, but since they can only act at the time of compilation they are not applicable to interaction control, where the control structures are changing during an interaction.

Functional languages offer an interesting alternative, partly because of the self optimising mechanisms implied by combinator manipulations, and partly because such code is referentially transparent, and therefore safely managed by optimisation activities. One must not forget though that a continuous interactive system is basically transforming infinite input streams into infinite output streams, and thus lazy evaluation of those streams is a fundamental requirement.

So the current issue of significance to interaction design is that of *control expression.* The designer needs to be able to efficiently determine the flow of control and such matters cannot be delegated to a general purpose language and compiler. For instance, an interface designed in Prolog often contains the frequent use of "cut" to limit the backtracking, because it is not relevant to the task, and this is an indication of an inappropriate language for the task. On the other hand, the complete lack of high level control expressions in a conventional language allows the designer to create many different control strategies, with the result that the program becomes more opaque and difficult to maintain. Our requirement is therefore to be able to declare control strategies, which are easily understood by the designer, and yet have a degree of adaptability to the task, during execution.

3.3.2 Data structures for control: The data structures which are required to represent the interaction states are themselves more complex than a simple hierarchy, with the simplest extension being that of allowing parts of the

structure to cross refer to each other. An example of such cross linkage is the need to express, in a visual image, that two images must be coordinated, i.e. change in one should change the other, since they are both representing the same thing. Traditionally, such coordination has been contained in an "application program" which in turn modified an external data structure, which in turn redrew the picture. In contrast, our proposition is that the data structure should contain both the definition of the image and the coordinating logical pointers. In other words, those portions of an application which control visual interaction should be delegated to a purpose made interaction controller and expressed in an appropriate language.

A more general view of the interaction data structure is that each item in it is allowed to contain multiple attributes, some visible and some not, which can be grouped with different hierarchies according to the class of attribute. Such data structures are known in databases, but remember we are defining the *interaction control structure*, which is at the heart of an interactive interface, where performance and flexibility are more important than size. LOOPS provides an interesting example of this type of complexity, with the "active value" mechanism, which could be used to maintain a co-relation between two objects. However, a system with many such links could become unmanageable.

An important feature of the data structure is that access to it should be constant for all items in the structure. This implies that one should not build it with list structures, which require sequential access down a list. Instead, random access to elements of the structure is needed, which in turn implies the dynamic management of access paths. This may conflict with the ideas of functional programming, which to date has relied heavily on list concepts to build data structures, and this matter needs careful investigation.

Thus, to summarise, the task of interaction design requires a declarative environment which can efficiently manipulate and search multi-dimensional structures. Safe re-usability of code is essential to reduce the large volume of design, and ideally the system itself should be able to compose new varieties of interface automatically as it adapts to the user. The designer should be able to design by expression of constraints and functions, and should be able to consult predefined bodies of expertise on presentation technique.

3.4 Efficiency and performance

Naturally, all the above generality is required without sacrifice of interactive performance. There is ample evidence of the importance of response times in the usability of a system. Some research even suggests that with a response time of less than a third of a second the performance of skilled users makes a significant improvement. Assuming this to be so, how can we afford the richness of data and control structure discussed above? Sometimes the simplest of actions of users can cause significant rearrangement of the structure and delay of response is most unacceptable.

We have experimented with some of the ideas outlined above, on a PERQ, and found the performance to be inadequate. (Naturally, we could have removed some of the generality and optimised the code, but the purpose of the experiment was to find tools which reduce programming effort, not increase it.) The latest workstations based on the 68020 should be better, but the power will still probably be borderline. Given that the time is consumed on pattern matching and structural manipulation, the use of current VLSI graphic controllers will not contribute a great deal either.

There is a possibility that "lazy evaluation" of the result of actions will help contain the visible performance within acceptable bounds. In the case of selecting items on a screen (picking), the search space is restricted by visible extent of the data structure. In general, the consequential changes resulting from an action need to be scheduled according to a precomputed dependency graph so that the fastest possible reaction is provided. It is also important to schedule action at level of detail which has the result of saving execution time. For instance, there is generally little value in deciding whether or not to redraw an individual line, since the time needed to draw the line is small compared to the time needed to evaluate the choice. On the other hand, the redrawing of a large set of lines is well worth evaluating.

So taking these requirements, of complex data structures and high level expression of search and control, all at high peak performance, it is highly likely that MMI can make use of many of the techniques being developed for the Fifth Generation, provided that a single user can afford their cost.

4 Conclusions

This document has attempted to review the aspects of human-computer systems from the point of view of users and particularly their resulting comprehension of their domain. It is not a simple story, and never will be, because humans are highly complicated and adaptable. It is important that we let computers evolve to be increasingly helpful tools and that we encourage humans to become expert masters.

Technology appears to be evolving in a useful direction, but the urgent need is to educate users to think carefully about the real nature of the problems they are tackling. In order to think better they need to make better use of appropriate languages, according to the domain they are comprehending.

LANGUAGES

Appropriate languages are needed in which knowledge can be expressed in a form that is convenient and natural for the user. These languages must be so structured that they can be transformed from high level statements down to efficient machine running code and are suitable for parallel execution. The Flagship architecture, for example, is based on declarative style programming.

Language-overview

E. Babb
ICL, Systems Strategy Centre, Bracknell, Berks, Great Britain

Abstract

A step back is made from the usual close view of just programming languages and consideration is given to a more general interpretation of computer language to include non-programming and engineering languages. Pure mathematical logic is one extreme method of communicating with a machine without program control. The other extreme is cell design languages used to design solutions in VLSI. Some brief conclusions are drawn about these different languages and what the future may hold.

1 Introduction

Computer languages form a spectrum going from the totally non-procedural based on mathematical logic to the engineering languages, such as those used to design VLSI. Database query languages are based on mathematical logic, whilst VLSI engineering involves detailed considerations of physical layout and behaviour of VLSI. Despite the attractions of the low levels to engineers, the end user doesn't want to be involved in these technicalities. He ideally wants to specify his problem in a familiar notation with the machine providing a solution at a reasonable cost. He only resorts to lower levels of language in desperation if the higher level does not provide him with sufficient performance.

On this spectrum *Pure Mathematical Logic* describes problems not methods and is *technology independent.* At the other extreme *VLSI engineering* describes how to get a piece of VLSI to perform operations such as arithmetic. It is therefore highly *technology dependent.* In general, the lower the level, the more options available to the designer as the following list illustrates:

TECHNOLOGY	DESIGN OPTIONS
Pure mathematics	No design options
Declarative languages	Instruction sequence
Programming languages	Store & instruction control
Circuit Engineering	Copper track layout
VLSI Engineering	VLSI layout and Physics

This increase in the number of options at the lower levels is the key to getting closer to optimal performance, but makes the design process more compli-

cated and therefore more expensive. Low level options also tend to interfere with global optimisation. This is the reason people prefer to use general purpose computers, rather than designing their own hardware.

2 Pure mathematical logic

Mathematical descriptions at their purest level aim to say nothing about how a problem is to be solved. They are a method of stating a problem in a manner that is technology independent. At the human level they match natural language much better than any procedural language. Even designers drawing electronic circuits, factory layouts or mechanical structures are in reality writing the equivalent of a large set of differential equations plus rules in logic. They would like these designs to come alive. Currently, they rely on intuition, rules of thumb and the occasional computer program to provide partial animation.

2.1 An example

Here is a request for information on *the parts that weigh more than 3 pounds.* In Logic it would look like:

query: $\exists$ *(weight) (weight* $>$ *3 &* $<$*part, weight* $>\in$*part-weight))*

answer: part = 'p1 or part = 'p2

The answer is all the part substitutions (p1 and p2) that would make the query true. The query is technology independent because it makes no attempt to say how the machine should solve the problem. Thus it is quite distinct from the search algorithms or hardware that are designed for fast calculation. Because of the technology independence, the user can concentrate on specifying his problem rather than worrying about a particular solution technique.

2.2 Executing logic

Executing logic is best done by representing all mathematics in a formal logic such as predicate calculus. To execute the above example:

query: $\exists$ *(weight) (weight* $>$ *3 &* $<$*part, weight* $>\in$*part-weight))*

we eliminate the existential quantifier $\exists$ *(weight)* by making *weight* a local variable:

query: local-weight $>$ *3 &* $<$*part, local-weight* $>\in$*part-weight)*

We then try to execute *local-weight* $>$ *3* but trap the fact that an infinite set of substitutions (*e.g. 4,5,6,...*) would make this true – and it is therefore non-

terminating. Provided we have managed to trap this non-termination we can employ a suitable equivalence theorem to change to an equivalent terminating formula:

query: $<part, local\text{-}weight> \in part\text{-}weight) \ \& \ local\text{-}weight > 3$

The term $<part, local\text{-}weight> \in part\text{-}weight)$ when mapped to a suitable algorithm generates a finite set of substitutions which become input to the now finite test $local\text{-}weight > 3$.

Thus the execution of logic involves:

1 A set of termination theorems which can detect non-termination of a problem or sub-problem prior to execution.
2 A set of transform-theorems to transform our non-terminating problem to equivalent terminating sub-problems.
3 A set of basic algorithms which can solve these sub-problems quickly.

Usually all this is done by designers and engineers or alternatively as with database and knowledge systems by specialised end user packages.

3 Declarative languages

3.1 Functional languages

Functional languages give more control because usually the functions are executed as written. Therefore, because we execute the equations in a fixed manner, the programmer must make sure the statements are ordered to give a reasonably fast algorithm. These languages often have very little control over storage which is usually dynamically allocated as required. Some functional languages provide no destructive assignment and so the opportunity for the programmer to control the allocation of storage almost disappears and depends on the cleverness of the compiler. Some of these languages have no type checking (i.e. declaring variables to belong to some general class such as the set of integers), the languages ML and Common Lisp being exceptions.

Example: suppose we are given a function *f-weight-part* that generates weight-part pairs. From these pairs we must filter those with weight less than 3. Therefore we can ask the earlier query in functional notation:

query: (f-filter-greater-3 (f-weight-part))

answer: '(p1 p2)

Functional languages usually use fixed computation rules and so queries can be difficult to modify.

3.2 Logic programming languages

Logic languages such as PROLOG currently use deterministic execution strategies such as left to right and so are algorithmic. Thus if we attempt to execute the following in PROLOG an error will occur because the term *weight* > *3* cannot finitely generate all the bindings for the variable *weight:*

query: weight > *3* & < *part, weight* > ∈*part-weight*

By manually reordering the terms a terminating algorithm is obtained:

better query: < *part, weight* > ∈*part-weight* & *weight* > *3*

PROLOG occupies an important position below a pure logic language and provides a good way of specifying non-numerical algorithms where performance is not of prime concern.

3.3 Combined logic and functional language

There are times when the functional notation is best, particularly for scientific computation, and other times, as in the earlier parts example, where logic notation is preferred. Ideally they both should be available and map to a common internal form perhaps along the lines being pursued in Hope.

4 Programming languages

Traditional programming languages such as C, Fortran, Pascal, COBOL, etc. essentially aim to provide almost direct control over the computer but in a friendly notation. They may include type checking of variables to provide a crude validation that all is well. Assembler level languages aim to provide the full power of the machine to the user. Sometimes they can be easier to use than high level languages because the semantics of each instruction is more precisely defined. However, in general this is not the case, since the programmer may be expected to deal with different levels of physical store and other details.

5 Circuit engineering

Design of the printed circuit board provides further scope for improving machine speed. At this level parallelism appears in the design language, since many gates are operating in parallel. There are many examples where ICL designers have identified critical algorithms and moved them into special hardware. For example CAFS is the hardware embodiment of search and selection algorithm, whilst the DAP originally embodied algorithms for solving differential equations and matrix manipulation.

6 VLSI engineering

This level of design offers considerable flexibility via silicon compilers or cell compilers. These offer the ability to define the behaviour, physical arrangement,

structure and geometry of a piece of VLSI. The behavioural definition says what should happen in terms of input output relationships. The structural representation links the behavioural description to the geometrical definition. At the geometrical and physical level, knowledge of power distribution, block interconnection and packaging must all be used in the design process.

7 Conclusion

As these sections show, lower levels of design, such as VLSI, require more effort because of the larger number of design options available. Of course, by using these lower levels we could get massive performance benefits. However, designers of end user systems usually carefully balance the cost of this design effort against the cost benefits of extra performance. Usually only if an end user artifact is to be used by large numbers of people does a large low level design effort make economic sense.

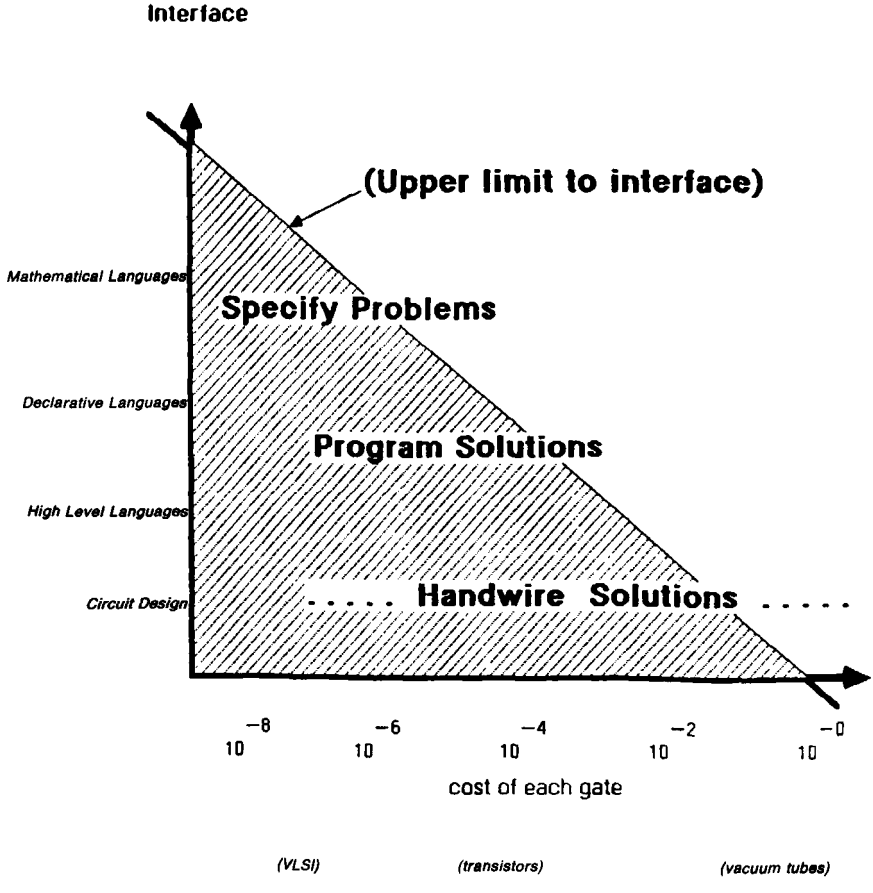


Fig. 1 Gate cost – language level

The coarse options available to a designer are summarised by Fig. 1. We can see that VLSI, with its low gate cost, whilst still sustaining "hand wired" solutions for complete CPUs or special graphics functions, is not usually used directly to build end user systems. Using cheap VLSI gates, the trend is toward using Mathematical Logic, in some friendly form, to build our end user systems – actually making it easy for end users to build their own system. Contrast this with vacuum tubes, with their high gate cost, where end user systems were actually built by special hand wiring. Thus in those early days, as the figure shows, the upper economic interface limit was very low; now it is potentially very high.

Now that the Pure Logic level is becoming economic we can imagine many of the operations required by such an interface being performed by special VLSI architectures – effectively giving a further cost reduction than just a crude gate cost reduction would imply. So we see that, just as the viability of Logic is fuelled by cheap VLSI gates, so Logic will fuel the production of special VLSI technology to help execute Logic operations.

Logic is not just an end in itself; it offers the advantage of closeness to natural communication, with its emphasis on problems rather than solutions. It also offers the crucial technical advantage of being able to reason easily about itself and any algorithms used. It is this ability to reason, especially about algorithms, that may distinguish the next generation of computer systems. It may be the most significant technological advance in the next decade.

PISA – A Persistent Information Space Architecture

Malcolm Atkinson
Department of Computing Science, University of Glasgow
Ron Morrison
Department of Computational Science, University of St Andrews
Graham Pratten
STL North West, STC Technology Ltd, Newcastle under Lyme, Staffordshire

Abstract

This paper describes the work of the PISA Project. The project is concerned with the development of Persistent Languages and the Persistent Information Space Architectures which support these Persistent Languages. The Persistent Languages are called persistent because they provide a consistent language view of all data and functions of whatever persistence, including short lived data traditionally held in program workspaces or appearing at human computer interfaces and long lived data traditionally held in databases. Persistent Languages are now developing beyond this initial aim. They will ultimately provide application developers with a simple and consistent view of all relevant system capabilities including concurrency, sharing and distribution. They will insulate application developers from changes in the implementation of these underlying capabilities and allow independent and easier evolution of applications and underlying capabilities. This paper decribes the PISA Project's approach to the design of Persistent Languages and Persistent Information Space Architectures and its relationship with other projects, in particular Flagship and IPSE 2.5.

1 Introduction

This paper describes the work of the PISA Project, a collaborative project funded by Alvey, ICL and SERC and involving the University of Glasgow, the University of St Andrews and STL North West.

If we examine existing computer systems we find dichotomies in their design and in that of their user interfaces. Some of these dichotomies were deliberately introduced during the 1960s and 1970s to satisfy perfectly reasonable performance constraints. Others are there simply because they reflect the order in which the basic design concepts of computer systems emerged over a period of twenty or thirty years. In referring to the

dichotomies in existing computer systems we have in mind those between languages and interfaces, programming languages and database languages, database languages and human computer interfaces (hci), virtual memories and databases, workspaces and databases, programming languages and operating system interfaces, etc. We contend that these dichotomies make the development and use of computer systems much more costly than they should be. They greatly increase the cost of evolving and enhancing the computer systems and the cost of developing applications based on them. They are seriously delaying the development of the Knowledge Based Systems and Integrated Project Support Environments of the future.

Because of the dramatic shift in the relative costs of hardware and software it has now become both feasible and timely for us to reconsider these dichotomies and remove those that cannot be justified. This has provided the broad context for the PISA Project. Within this broad context the project has started by considering the dichotomy between programming languages and database languages and has then gone on to consider the dichotomy between programming languages and hci interfaces. Application developers are now beginning to develop applications which store hci images (and the functions which generate these images) in databases. So the PISA project is also concerning itself with that third dichotomy, the dichotomy between database languages and hci interfaces. It has been estimated that in any application program only ten per cent of the program is concerned with the problem the application is trying to solve: the remainder is concerned with the conversion between the different views of data supported by programming languages, database languages and hci interfaces. The aim of the PISA Project is to investigate ways of reducing this overhead.

The PISA Project's approach is to develop Persistent Languages. These languages provide the same view of objects irrespective of whether the objects are held in a program's workspace, or are held in a database, or are seen at an hci interface. By our use of the term 'objects' we mean functions, procedures, etc. as well as data. By our use of the expression 'same view of objects' we mean the rules for constructing the objects out of other objects and the sets of operations which can be applied to the objects. The languages are called Persistent Languages because they provide the same view of objects irrespective of whether the objects exist (or persist) for a long time (i.e. objects normally held in a database) or exist (or persist) for a short time (i.e. objects normally held in a program's workspace or seen at the hci interface). A program written in a Persistent Language does not have the workspace and the interfaces to databases and hci possessed by a program written in a conventional language. Instead it exists in the context of one Persistent Information Space which contains all objects accessed by all programs. The Persistent Information Space Architecture which supports the Persistent Information Space is responsible for the secure storage of the long-lived parts of the Persistent Information Space, for the display at the hci interface of relevant parts of the Persistent Information Space and for the

control of concurrent access to the Persistent Information Space by many processes and many human users.

It might seem from the above discourse that the objective of the PISA project is to produce one super, general, all-purpose language, à la PL1 or Ada. This is not so. The project accepts that application developers need different languages in order to reflect the needs of different types of application (although for the purposes of its research the project will only investigate one or two languages). What the project does not accept is that application developers need different types of language to reflect the needs of different types of support system, i.e. database system, operating system, etc. So the PISA Project is investigating how we can design and support languages which reflect the needs of application developers rather than the needs, idiosyncrasies and divisions in the systems which support the languages.

There has been considerable reference recently to the desirability of 'object oriented language systems'. This interest began in the late 1960s with the development of the Simula language system and was greatly stimulated in the 1970s by the development of the Smalltalk system and in the 1980s by the creation of Lisp based, object oriented, systems such as Loops and Flavors. The aim of object oriented language systems is the same as the aim of the PISA Project: to provide one coherent homogeneous view of all objects which are of interest to the application developer. Although there has been much interest in object oriented language systems it has become apparent recently that the meaning of the term 'object oriented' varies considerably from system to system. The variations depend on issues such as granularity of objects, range of objects, typing of objects, etc. These are the issues which are of concern to the PISA Project.

2 The issues of concern to the PISA Project

In developing the very simple Persistent Language and Persistent Information Space image for the application developer the PISA Project has had to deal with a range of issues. Some of these have already been tackled by the PISA Project in developing the PS-algol Persistent Language system, available on 3900, VAX, SUN, Perq and Apple Macintosh hardware and on VME and Unix operating systems. Other issues have been tackled in experimental extensions to the PS-algol language, in a major development from the PS-algol language known as the Napier language, and in various design proposals. Some issues will be tackled in later stages of the PISA Project; some have been identified by the PISA Project but are considered to be beyond its scope.

2.1 Host and guest languages

When an application developer uses a conventional programming language system to develop an application he does not simply use the programming language supported by the system. He also uses the set of procedures which have already been implemented in the language. We could think of the

programming language as the 'host' language of the system and the programming language together with the set of procedures as a 'guest' language. It is the guest languages which are used by the application developers in developing their applications. The procedure call mechanism provides the extensibility feature in the host language which allows guest languages to be developed. These guest languages have proved the Achilles heel of the application development process. They have not been thought of as languages, have not been designed as languages and have been allowed to become incoherent and cumbersome.

In principle this situation has not been changed by the advent of Persistent Languages. Although the Persistent Languages now provide the host languages for application developers, guest languages still have to be developed on top of these host languages. However by their very nature the persistence capabilities in the new host languages must simplify at least some of the features of guest languages. In particular they obviously simplify those features which are concerned with access to resources such as databases, hci, etc. In addition the new host languages provide much more powerful bases for the development of guest languages than their predecessors. The new host languages do not simply provide the limited extensibility features of earlier languages, they provide the type algebra and abstract data type mechanisms discussed below. The guest languages can be built using these features.

We are providing a host language for the application developer (PS-algol now, Napier later) as part of the development of the PISA system. We are also providing the lowest level of guest language. In developing the PISA System we are faced with the choice as to which features should be included in the host language and which features should be included in guest languages. In making this choice we can appeal to a number of guidelines. If a system feature (say distribution) can be made entirely transparent to the application developer or if it needs to appear the same to all application developers then we can deal with that feature in the host language. If a system feature (say configuration management) needs to be protected from abuse by the application developer then we should deal with that feature in the host language or in the lowest levels of guest language. If we feel that the exact presentation to the application developer of a particular feature (say concurrency or interfaces to existing databases) needs further research or might vary from application developer to application developer then we can provide that feature in its simplest form in the host language and in a more powerful form in guest languages.

2.2 Naming view of objects

In our definition of Persistent Languages in section 1 we stated that Persistent Languages provide the same view of objects irrespective of whether the objects exist (or persist) for a long time (i.e. objects normally held in a database) or exist (or persist) for a short time (i.e. objects normally held in a program's workspace or seen at the hci). We now need to consider in

more detail what we mean by the expression 'same view of objects'. The view of objects seen via Persistent Languages is largely determined by the naming system of the language which we will discuss in this section and the type system which we will discuss in section 2.4.

Consider first the view of objects which is seen by a procedure written in a conventional language. The procedure sees and accesses objects via three mechanisms. Firstly there are objects (e.g. local variables) which are created when the procedure is invoked and which are destroyed when exit is made from the procedure. Secondly there are objects which are passed as parameters to the procedure when the procedure is invoked or which are returned as results by the procedure when exit is made from the procedure. An object passed as a parameter to the procedure may be modified in some way by the execution of the procedure (e.g. the value of a variable may be updated). If the object is modified then it is the modified version of the object which persists after exit is made from the procedure, not the original version. (We are talking here of languages with store semantics and not of applicative languages.) Thirdly there are files or databases held in the file system; the procedure always accesses these files or databases by invoking special sets of procedures such as created, read, write, etc. These special sets of procedures are said to encapsulate the files or databases because they provide surrounding interfaces which must be used when accessing the files or databases.

Now assume that an application developer wants to implement an application procedure which accesses a large structured collection of objects. Given the scenario presented in the previous paragraph the application developer would expect to be able to use either the second or the third mechanism to gain access to this structure from the application procedure. If he used the second mechanism then, when the application procedure was invoked, a pointer to the structure would be passed as a parameter to the procedure. The formal name of the parameter within the procedure would effectively be the local name of this pointer. The procedure could then select objects within the structure just as it selects elements within an array. It could assign these objects within the structure as values to other variables. The names of the variables would then effectively provide local names for pointers into the total structure. Thus using the normal language mechanisms of selection and assignment the procedure would be able to select any object within the structure. We could say that it could use these mechanisms to navigate its way around the structure just as it can use the interface to a conventional network database system to navigate its way around a network database. This process of navigation using selection and assignment provides the procedure with a contextual naming mechanism which enables it to reach any object within the total structure. Objects can only be selected or named in the context of the objects containing them within the structure. Objects do not need to be provided with names which are unique within this total structure. Having reached any object within the structure using the contextual naming mechanism the procedure can use the assignment mechanism to modify that object.

However with a conventional language the second mechanism can only be used for accessing objects held in the workspace of an application, i.e. the objects which exist only while the application is running. So if the application developer wants to implement an application procedure which accesses a large structured permanent (or long lived) collection of objects he is forced to use the third mechanism described above, even though the second mechanism may be more appropriate.

Persistent Languages still support the three mechanisms described above. However they allow the second mechanism to be used for accessing long lived or permanent data. So any part of the total Persistent Information Space can be accessed using the second mechanism. The third mechanism is therefore not required any more as a mechanism for encapsulating permanent data held in files and databases. Instead it is seen as a mechanism for building new views or abstractions on top of the objects available in the Persistent Information Space, i.e. as a mechanism for building abstract objects, i.e. the mechanism known as the Abstract Data Type (ADT) mechanism. In some circumstances the abstract objects will be required because they provide richer, simpler or just different views of the underlying objects in the Persistent Information Space. In other circumstances they will be required because they provide privacy checks which control access to the underlying objects. The third mechanism is in fact built on top of the second mechanism. The procedures which encapsulate the abstract objects supported by the third mechanism are themselves objects supported by the second mechanism.

The discourse above on the second access mechanism described the essentials of the contextual naming mechanism provided in conventional and Persistent Languages. This mechanism was described as the means by which a procedure could navigate to objects within a large structure of objects. The same mechanism can now be used as a means by which a procedure implemented in a Persistent Language can navigate to objects within the total Persistent Information Space. So the scope and usefulness of the second mechanism has been increased enormously.

The problems of configuration management and version handling are closely related to the problems of contextual naming and typing. We expect solutions to these problems to make use of the capabilities described here and in section 2.4. However these problems are not officially within the scope of the PISA Project.

In this section we have been discussing the interaction between a procedure written in a Persistent Language and the run time environment of that procedure as provided by the Persistent Information Space. We have not talked about the interaction between the procedure and its compile time environment, i.e. the set of type declarations for objects used in the procedure. It is important to recognise that the introduction of Persistent Languages and Persistent Information Space Architectures has enabled us to

hold the compiler and the compile time environments of procedures in the same Persistent Information Space as the running versions of the procedures. Hence the Persistent Information Space Architecture provides a powerful base for the development of partial evaluation schemes which remove the sharp distinction between compilation and run time and replace it with incremental partial evaluation. Although partial evaluation was not within the original, official, objectives of the PISA project it has proved to be inseparable from those objectives.

2.3 Addressing objects and distribution

The previous section described the contextual naming mechanism as the means by which a procedure could navigate through the Persistent Information Space to the objects it requires. The routes through the Persistent Information Space to the objects provided the contexts for the objects within the space.

Now in fact the same object may appear in two contexts within the Persistent Information Space. This creates two problems. Firstly, how can an object in one context be incorporated into another context? Secondly, how do we recognise that two objects selected through two contexts are in fact the same object? The answer lies in the nature of the pointers referred to in the previous section.

If he likes, the user of the Persistent Language can think of these pointers as holding the unique addresses of the objects pointed at. In fact as the Persistent Information Space is unbounded in size we cannot have unique addresses as these would be unbounded in length. This is a problem for the implementer of the Persistent Information Space Architecture rather than for the user of the Persistent Languages. However we will describe briefly one approach to this addressing problem.

The store underlying the Persistent Information Space is organised into localities. A locality is normally the storage system of one machine. A pointer which points to an object in the same locality as the pointer is stored as an address within that locality. A pointer which points to an object in another locality is stored as an address within the other locality together with a means of communicating from the locality containing the pointer to that containing the object pointed at. Thus the pointer is effectively supported by a contextual addressing system. The pointer is unique within the addressing context formed by the locality holding the pointer and the set of communication links stemming from that locality but is not unique beyond that addressing context. Pointers have to be converted when they are moved from one addressing context to another, i.e. from one locality to another. This conversion is performed by the Persistent Information Space Architecture.

When a request is made in one location for an operation to be performed on an object in another location the system has a decision to make: should it

move the object to the location requesting the operation or should it move the software requesting and/or supporting the operation to the location containing the object? As both the objects to be operated on and the software requesting and supporting the operations are held in the same Persistent Information Space the choice between the two options is not as stark as it would be in a conventional system. Ideally the application developer should not be expected to make this choice; it should be made by the Persistent Information Space Architecture.

If the Persistent Information Space Architecture is to be truly effective it must be capable of supporting its Persistent Information Space on unbounded distributed heterogeneous storage systems because these are the storage systems available to application developers. If it is to do this it must provide a solution to some of the problems outlined in this section. Ideally the solutions to these problems should make the problems completely invisible to application developers, who should simply see the view of the Persistent Information Space described in sections 2.2 and 2.4. However, more research is required to fully achieve this objective.

2.4 Typing of objects

In early languages, such as Fortran and Algol, the type system of the language played a limited role. At compile time the user of the language could define the types of variables in terms of a limited set of base types (e.g. integers, reals,...) and a very limited and predefined set of type constructors (e.g. arrays,...). The compiler could then check the use of the variables in a program against the type definitions of the variables and could detect a limited set of program errors.

The theory of type systems and the type algebras they support has developed very rapidly over the last few years. The PISA project has developed the language Napier to incorporate the most advanced state-of-the-art type algebra into a Persistent Language. The type system of Napier includes conventional primitive types such as integer, real, boolean, string,.... It also includes primitive types suitable for use in hci interface designs, e.g. pixel, picture and image. It provides a rich set of type constructors such as vector, structure, union, procedure, abstract data type,.... The type constructors can be parameterised by type thus providing parametric polymorphism. This enables the application developer to define new type constructors such as set, stack, relation, etc. which can be applied to any type to create new types. The type system is capable of defining the types of functions, procedure interfaces, database structures and hci images as well as more conventional data types. The PS-algol Persistent Language is an older language than the Napier Persistent Language and therefore does not support the full type algebra capabilities of Napier. However it does support a very useful subset of these capabilities.

The type algebra is used in the support of the second access mechanism described in section 2.2. The Abstract Data Type (ADT) capability within

this type algebra is used in the support of the third access mechanism described in section 2.2.

The main objective of the developers of Persistent Languages has been to bring together in one language view objects normally accessed via a programming language and objects accessed via database languages. This implies that the type systems of programming languages and of database languages need to be brought together in the type systems of Persistent Languages. We believe that the type algebra and the ADT capability of Persistent Languages provide more than adequate capabilities for new databases. We also believe they can be used to simulate the interfaces to existing database systems.

2.5 First class persistent procedures

We stated in section 1 that Persistent Languages aim to provide a coherent homogeneous language view of all objects which are of interest to the application developer. By this we mean that they aim to treat all objects as 'first class citizens', that is to treat all objects in exactly the same way. They aim to allow each of the objects to be held in the Persistent Information Space, passed as a parameter, declared with a type declaration, etc. just like any other object.

Procedures are treated as first class citizens by the Persistent Languages and the Persistent Information Space Architecture. As we noted in section 2.2 this means that the procedures which are used to support the ADT mechanism can be held in the Persistent Information Space and treated like any other objects. Furthermore it means that the procedures which support program/procedure libraries can also be treated in this way. With this capability the existing PS-algol Persistent Language system is able to load procedures incrementally, type checking their bindings, when they are called by other procedures. This has two important implications. Firstly in a system built in this way the supplier does not need to implement and maintain separate library and linkage editing software. Secondly the application developer does not need to learn about, understand or explicitly use such software.

2.6 Concurrent sharing of objects

The existing PS-algol Persistent Language system has limited facilities for concurrent sharing of objects. It divides the Persistent Information Space up into databases, each of which can be concurrently accessed by many users in *read* mode or accessed by one user in *write* mode. Thus the capabilities of PS-algol databases are similar to those of Codasyl areas. The existing PS-algol Persistent Language system supports a one-level model of transactions based on implicit *start* and *abandon* transaction operations and an explicit *commit* transaction operation.

A transaction capability is useful even in a single user, single process system.

It enables the single process to have points within its operation where it decides whether or not to commit to changes in the Persistent Information Space made since it last committed. If it chooses not to commit then the state of the Persistent Information Space is rolled back to the state when the process last committed. One can even justify nested transactions in the context of a single user, single process system.

The approach of the PISA Project in its research on transaction capabilities has been to provide a very simple transaction capability in the Persistent Languages and Persistent Information Space Architectures and then to build more powerful capabilities on top of the Persistent Languages, i.e. in the guest languages referred to in section 2.2. We feel this approach is necessary at this stage because we observe that different application areas still require different transaction capabilities.

We believe that the model of concurrency eventually adopted by the PISA Project will be derived from the theories and languages being developed elsewhere in projects not concerned with the problems of persistence. Persistence will add a new dimension to these theories and languages because it will introduce the problems and benefits of persistent processes. Again we believe that the applications developers may require different flavours of concurrency. So the approach of the PISA Project in its research on concurrency has been to start with a minimum concurrency capability within the Persistent Languages and Persistent Information Space Architecture and experiment with richer capabilities at higher guest language levels.

2.7 Range of objects

Many systems claim to be object oriented. However when one examines these systems in detail one finds an enormous variation in the granularity and range of objects supported by the systems. In some of them the word 'object' is really used as a 'with it' alternative to the word 'file'. In these systems the granularity is coarse (i.e. effectively the file) and the range of objects is small (i.e. one object, the file). In language systems such as Smalltalk, Lisp/Loops and PS-algol the granularity is fine and the range of objects large because the latter includes any objects visible in the languages.

In choosing the range of objects which will be supported by its Persistent Languages and Persistent Information Space Architecture the PISA Project has been faced with an increasingly difficult sequence of decisions. As we said in section 2.5 the objective has been for the language and architecture to support each object in the range as a 'first class citizen'. Incorporating support for primitive data types such as integer, real, etc. and the set of type constructors for these types was a state-of-the-art activity. Incorporating hci objects such as pixel, etc. was also fairly straightforward although very beneficial. Incorporating functions and procedures was harder but again very beneficial. Treating processes, transactions, locks, etc. as first class citizens is the next hurdle. Treating types as objects with the same rights as any other

objects presents real theoretical and practical difficulties. The approach on the PISA Project has been to give the compiler special capabilities for handling types as objects and to withhold most of these capabilities from other software.

2.8 Multiple languages

The issues discussed above have been researched by the PISA Project in the context of the PS-algol and Napier Persistent Languages. If the research is to be widely exploited it will need to be applied in the context of the other languages which are now becoming of interest to application developers and software engineers, e.g. Smalltalk, Lisp, ML, and even C. However in order to do this effectively it may be necessary to make substantial changes to these languages, for instance in the area of type systems.

3 Relevance to other projects

There are two major Alvey Projects with which the PISA Project has close contact. These are the Flagship Project and the IPSE 2.5 Project.

3.1 The Flagship Project

Several papers in this issue deal with different aspects of this project; references are given at the end of this paper.

The core objective of the Flagship Project is to develop a parallel processor machine capable of efficiently supporting functional programming languages. Beyond this core objective the Flagship Project has the additional objectives of developing functional programming systems on its parallel processor machine and demonstrating that other declarative languages can also be supported on the machine.

However these objectives will only produce a parallel processor system providing efficient support for declarative language systems. If such systems are to achieve widespread commercial use an evolutionary path must be demonstrated which allows application developers to move to these systems from their existing application development systems, which must include fourth generation language systems, Lisp/Loops-like systems and IPSEs. It is this objective which takes Flagship beyond the objectives of a declarative language crunching engine development project.

In fulfilling this last objective the Flagship Project is adopting an approach consistent with that of the PISA Project. It is attempting to provide application developers with a Persistent Language view of the system capabilities they require, so for instance application developers using a functional language such as Hope will see the system capabilities they require through a Persistent Language extension to that language. The work on the Flagship PRM (Program Reference Model) has identified the set of system

capabilities which must be supported in this way in each Persistent Language supported by the Flagship machine.

The PISA Project is interfacing with the Flagship Project at three levels. Firstly the two projects are collaborating at the research level by exchanging ideas and research contacts on subjects such as type systems, storage systems, etc. This research collaboration has been particularly close in the context of the PRM design activity. Secondly Flagship is relying on PISA to help provide support on the prototype Flagship machine for Flagship's own Persistent Language (i.e. Hope plus PRM capabilities). It will do this using the version of the PISA Persistent Information Space Architecture currently available with the PS-algol Persistent Language system together with some interfacing glue written in PS-algol. Thirdly if the ideas of the PISA and Flagship Projects can be converged sufficiently then Flagship will be able to make use of all or part of the PISA Persistent Information Space Architecture in the final version of the Flagship machine.

3.2 The IPSE 2.5 Project

The mnemonic IPSE was coined by the Alvey Directorate to stand for Integrated Project Support Environments. It was envisaged that there would be three stages of IPSE: IPSE 1s comprising a set of autonomous tools interfacing with each other via a Unix-like file system, IPSE 2s comprising a set of tools integrated around a shared database, and IPSE 3s providing the ultimate knowledge base orientated IPSE. In the ultimate IPSE 3 the separation of tools and data would have completely disappeared as both would have been integrated into the knowledge base. The IPSE 2.5 Project was given the name IPSE 2.5 because it was seen as researching the steps going immediately beyond the state-of-the-art database-oriented IPSE 2s towards the ultimate IPSE 3s.

The objective of the IPSE 2.5 Project is to provide support for all stages of the software development process used in a software development project. At the core of an IPSE 2.5 system will be an information space containing a model of the total software development process. This will represent the subprocesses within this total process, the information used by these subprocesses and the flow of information between the subprocesses. The subprocesses represented will include tools and humans performing software development activities.

Ideally the IPSE 2.5 Project would not need to concern itself with the support of the information base at its core, but solely with the way the software development process was used and was represented in the information base. It would concern itself with the various subprocesses of the total software development process, such as formal methods, management support, etc.

So ideally the IPSE 2.5 Project would like to have available to it a Persistent Information Space Architecture and a Persistent Language to interface with

this architecture. It would like the Persistent Language to provide a language view of the system capabilities it requires. These capabilities would have to include a Persistent Information Space large enough to contain all information required by a software development project, a capabilility for storing processes as well as data in the Persistent Information Space, shared concurrent access to the Information Space by many humans and tools, security of the Information Space, etc.

So the aims of the PISA Project are highly compatible with the aims of the IPSE 2.5 Project. The only problem is the short timescales of both projects. At this point in time there are various language systems which are of interest to the IPSE 2.5 Project for historical and technical reasons. ML is of interest to the people concerned with the program-proving parts of the project, Meta IV and functional languages to the people working on formal methods, Smalltalk to those working on hci issues, Lisp/Loops to the project as a whole because of the wealth of AI tools built on them in the USA, PS-algol because it is already supported by a Persistent Information Space Architecture, etc. It is difficult in the timescales of the PISA and IPSE 2.5 Projects to bring all of these language systems together with the Persistent Language approach. However this reconciliation will be needed if we are to move forward on a longterm IPSE development path.

3.3 Other projects

Space does not permit us to consider in detail all projects which have interests in common with PISA. However in an ICL context we should be considering the relevance of PISA research to ICL's database strategy, application architecture, mainframe architecture and standards activities. The relationships of the PISA Project to the PCTE and ANSA Projects is important because these projects are being used as standards foci within Europe and Britain.

The aim of the PISA Project is to provide application developers with a coherent homogeneous language view of the objects they require, in particular the objects traditionally supported by language systems, database systems and hci. It must initially provide this view on top of the heterogeneous collection of storage, processing and communication mechanisms available at present. The effect of this work should be to insulate or decouple the application developer from these underlying storage, processing and communication mechanisms. This should in turn allow more flexibility in the development of both applications and underlying mechanisms.

Acknowledgments

We wish to acknowledge the financial support provided by Alvey, ICL and SERC and the encouragement and interest we have received from individuals in those organisations.

We wish to acknowledge the work of the members of the PISA Project team, who have produced the existing versions of the PISA system. They have also produced numerous papers describing various facets of the PISA Persistent Languages and the Persistent Information Space Architecture. A list of these papers can be obtained from the manager of the PISA Project, Nick Capon, STL North West, STC Technology Ltd, STC.

References

A The following papers introduce the IPSE 2.5 and Flagship Projects:

1 SNOWDON, R.A.: 'Advanced Support Environment Study Final Alvey Report'. ICL/STC.

2 TOWNSEND, P.: 'Flagship Hardware and Implementation'. ICLTJ, 1987 Vol. 5, No. 3, 575–594.

B The following papers add to the limited outline of the PISA Project given in this paper or represent the evolution of ideas in the project:

1 ATKINSON, M.P., MORRISON, R. and PRATTEN, G.D.: 'A Persistent Information Space Architecture'. Dublin, 1986.

2 MORRISON, R., DEARLE, A., BROWN, A. and ATKINSON, M.P.: 'An Integrated Graphics Programming Environment'. Computer Graphics Forum, 1986, Vol. 5, No. 2, 147–157.

3 ATKINSON, M.P. and MORRISON, R.: 'Integrated Persistent Programming Systems'. 19th Annual Hawaii International Conference on System Sciences, 1986, Vol. IIA, Western Periodicals Co.

4 ATKINSON, M.P. and MORRISON, R.: 'Procedures as Persistent Data Objects'. ACM TOPLAS, 1985, Vol. 7, No. 4, 539–559.

5 COCKSHOTT, W.P., ATKINSON, M.P., CHISHOLM, K.J., BAILEY, P.J. and MORRISON, R.: 'POMS: A Persistent Object Management System'. Software Practice and Experience, 1984, Vol. 14, No. 1, 49–71.

6 ATKINSON, M.P., BAILEY, P.J., CHICHOLM, K.J., COCKSHOTT, W.P. and MORRISON, R.: 'An Approach to Persistent Programming'. Computer Journal, 1983, Vol. 26, No. 4, 360–365.

7 ATKINSON, M.P.: 'Programming Languages and Databases'. Proceedings of the 4th International Conference on Very Large Databases, Berlin, IEEE, 1978, 408–419.

C The following papers are some of the papers which have provided technical inspiration to the PISA Project:

1 CARDELLI, L. and WEGNER, P.: 'On Understanding Types, Data Abstraction and Polymorphism'. ACM Surveys, 1986.

2 WEGNER, P.: 'Language Paradigms for Programming in the Large'. St Andrews University Easter Lecture Course.

3 ALBANO, A., CARDELLI, L. and ORSINI, R.: 'Galileo, a Strongly Typed Interactive Conceptual Language'. ACM Transactions on Database Systems, 1985, Vol. 10, No. 2, 230–260.

4 BURSTALL, R. and LAMPSON, B.: 'A Kernal Language for Abstract Data Types and Modules'. Proc. Int. Symposium Semantics of Data Types, 1984.

5 LISKOV et al: 'Preliminary ARGUS Reference Manual'. Programming Methodology Group, MIT, 1983.

6 ICHBIAH et al: 'The Programming Language Ada Reference Manual'.

7 SCHMIDT, J.W.: 'Some High Level Language Constructs for Data of Type Relation'. ACM Transactions on Database Systems, 1977, Vol. 12, No. 3, 247–281.

8 TENNANT, R.D.: 'Language Design Methods Based on Semantic Principles'. Acta Informatics 8, 1977, 97–112.

9 HOARE, C.A.R.: 'Monitors: An Operating System Structuring Concept'. CACM, 1974, Vol. 17, No. 10, 549–557.

10 ARDEN, B.W., GALLER, B.A., O'BRIEN, T.C. and WESTERVELT, F.H.: 'Program and Addressing Structure in a Time-Sharing Environment'. JACM, 1966, Vol. 13, No. 1.
11 DENNIS, J.R.: 'Segmentation and the Design of Multiprogrammed Computer Systems'. JACM, 1985, Vol. 12, No. 4, 589–602.

A more conplete list of references can be obtained from the manager of the PISA Project.

Software development using functional programming languages

J. Darlington
Department of Computing, Imperial College of Science and Technology

Abstract

Functional programming languages offer radical solutions to many of the problems currently met in software development and maintenance and if adopted could lead to dramatically different programming methodologies. This paper introduces some of the ideas behind functional programming languages and outlines the software technology research being undertaken within the Flagship project at Imperial College.

1 Declarative languages

The heart of any Fifth Generation project involves the adoption of one or more **declarative languages** and their associated software and hardware technologies. Declarative languages represent a radical departure from the conventional languages in widespread use today such as Pascal or Ada. Such conventional or **imperative** languages are intimately tied to the von Neumann model of computation which, it is claimed, imposes serious limitations on their effectiveness both in terms of software productivity and execution efficiency.

A major task of a programmer using a conventional language is to organise a linear sequence of side effect-inducing operations to achieve the desired overall effect. This style of programming has several undesirable consequences.

(i) The organisation of operations in the correct temporal sequence is very burdensome to the programmer. Programming is thus tedious and error prone and the resulting programs are unnecessarily complicated and verbose.

(ii) The resulting languages are not amenable to conventional mathematical manipulations. Programming is therefore carried out via an unscientific process of testing and debugging. As programmers we very much lack the ability to build formal models and prove that our solutions are correct before testing which can be found in more mature engineering disciplines.

(iii) Because the languages used and the underlying model of computation are inherently sequential it is very difficult to achieve improvements in execution efficiency by employing **parallel** or **concurrent** evaluation.

In contrast the declarative languages trace their origin to mathematical formalisms developed independently of any computing machinery. They therefore inherit the desirable properties of such formalisms and avoid the limitations listed above.

(i) The task of a programmer using a declarative language is to denote which values should be computed rather than organise the computation to produce the answer. Thus the languages are more concise and expressive and the burden to the programmer is much less.
(ii) By their nature the declarative languages are mathematically tractable. Thus the rigour and accuracy of mathematics can be applied to the programming process with the potential for large improvements in programmer accuracy and productivity. As many of the operations involved in programming can now be formally and concretely expressed there is the possibility of **automating** much more of the program development and maintenance process.
(iii) As declarative language programs do not require a precise evaluation order parallel evaluation is possible.

Within the declarative languages, at the moment, there are two main schools. The **functional languages** are based on the lambda calculus and equational systems while the **logic programming languages** are based on the First Order Predicate Calculus. Both styles of languages share the basic benefits of being declarative listed above but there are important differences between them.

By our definition, Flagship is a classic Fifth Generation project, aiming to develop the software and hardware technology made possible by declarative languages to a point where they can support commercial application development. In this paper we will introduce the software technology work being pursued by the Functional Programming Group in the Department of Computing at Imperial College as our main contribution to Flagship.

2 Introduction to functional programming languages

The Functional Programming Group at Imperial College has centred its work around the functional language Hope. Hope was first designed and implemented at Edinburgh University[1] and is a good representative of a modern functional language.

A Hope program is a set of equations defining functions. Separate equations can be written for separate cases of the input variable. For example

```
--- fib(0) <=1
--- fib(1) <=1
--- fib(n+2) <= fib(n+1)+fib(n)
```

defines the Fibonacci numbers.

Hope is strongly typed. Before being defined a function must have its type declared; this is achieved using the *dec* statement

```
dec fib:num->num
```

Hope employs polymorphic type checking so that type declarations can involve type variables, e.g.

```
typevar alpha
dec f :alpha->num
--- f(a) <= 0
```

is the stubborn function that returns 0 whatever you give it. Data structures in Hope are represented as terms built up from *constructor* functions, i.e. functions having no equations. These are introduced using the data statement. Thus

```
data listnum == nil ++ cons(num # listnum)
```

defines the data type *list of numbers* built up using the constructors *nil* (the empty list) and *cons*. Data statements can also be parameterised thus

```
data list(alpha) == nil ++ cons(alpha # list(alpha))
```

now defines a type constructor *list* such that list(num) is equivalent to the type listnum defined earlier.

Thus

```
dec length:list(alpha)->num
--- length(nil) <= 0
--- length(cons(a,ℓ)) <= 1+length(ℓ)
```

calculates the length of any list whatever its components.

Hope allows the normal shorthand for lists thus [1,2,3] is shorthand for cons (1,cons(2,cons(3,nil)))

Running a Hope program involves reducing an expression until it is totally composed of constructor functions, i.e. no more equations apply so length([1,2,3]) reduces to 3.

Infix operators are widely used in Hope, thus we can define :: as an infix operator for cons and the above equations become

```
--- length(nil) <= 0
--- length(a::ℓ) <= 1+length(ℓ)
```

Being higher order Hope allows functions to be passed as parameters and returned as values. Thus

```
typevar alpha, beta
dec * : (alpha->beta) # list(alpha)->list(beta)
infix * : 6
--- f * nil <= nil
--- f * (a::ℓ) <= f(a) :: (f * ℓ)
```

defines an operator * that applies a function to every element of a list, thus

fact * [1,2,3] evaluates to [1,2,6]

Higher order functions are especially useful as they allow many recursive definitions to be rendered by a single function application. Of particular use are functions over sets and Hope has borrowed the traditional set comprehension schema, thus

```
primesquares: set(num)->set(num).
primesquares(S) <= {n²|n in S : isprime(n)}
```

is the set of squares of all primes contained in a given set.

3 Flagship software developments

The software work being undertaken for Flagship in the Functional Programming Section at Imperial College has two main aims: the development of a more powerful and commercially viable functional programming language and the design and construction of a prototype program development and maintenance system based on the ideas of formal correctness-preserving program transformations.

Within the language area there are three main axes along which research is progressing

(i) Language expressibility. Staying within the basic functional framework we are seeking ways to increase the expressive power and applicability of the languages. Two ways of achieving this, incorporation of logic programming capability and temporal synchronisation, are examined in more detail below in 4.1 and 4.2.

(ii) System architecture. To date it has perhaps been a legitimate criticism that the majority of functional languages are fine in isolation but do not interact well with the rest of the non-functional world. Within Flagship we are seeking to rectify this by developing a more sophisticated system architecture and expressing the capabilities thus provided via the language. In particular we are incorporating into the language:
   (a) Persistence. The view that any object created during computation persists and can be accessed as long as it is named fits very nicely

with the functional view and is actually easier to accommodate than the current character-based input/output systems.

(b) Language interworking. The ability to use procedures written in other languages will greatly extend the useability and acceptability of functional languages.

(iii) Program forms and programming in the large. Functional language research abounds with innovative ideas, particularly in the areas affecting the static semantics of the languages, e.g. typing regimes and module structures. These will be evaluated and incorporated into the new language if they prove practical, particularly when applied to large scale applications.

4 Language developments

4.1 Logic programming extensions of functional languages

A comparison between logic programming and functional programming reveals that languages from the former camp possess two attributes not available to functional programmers. Firstly because a logic program makes no commitment as to which variables in a relation are considered as inputs and which as outputs, a logic relation once defined can be used in several different **modes**. For example given the *append* **relation** defined in Prolog thus

append(nil, ℓ, ℓ)
append(x :: ℓ1, ℓ2, x :: ℓ3) :- append(ℓ1, ℓ2, ℓ3)

it can be used in a 'functional' way to join lists together by means of a goal statement such as

append([1, 2], [3, 4], ℓ)

which will succeed binding ℓ to [1, 2, 3, 4]. But it can also be used 'backwards' via a goal statement such as

append(ℓ1, ℓ2, [1, 2, 3, 4])

which will produce all the values for ℓ1 and ℓ2 that when appended give [1, 2, 3, 4] viz.

[], [1, 2, 3, 4]
and [1], [2, 3, 4]
and [1, 2], [3, 4]
and [1, 2, 3], [4]
and [1, 2, 3, 4], []

In contrast, functional programs are committed as to what they regard as inputs and what they regard as outputs. Consequently the append **function** cannot be used to split a list as above; a separate function would have to be defined.

Another capability found in logic programming but not in functional programming is that the results produced need not be totally **ground**, i.e. they may involve variables. For example given the length relation defined thus

```
length(nil, 0)
length(x :: ℓ, n1) :- length(ℓ, n), plus(n, 1, n1)
```

a query of the form

```
length(ℓ, 2)
```

would succeed producing a binding for ℓ of the form u1 :: (u2 :: nil) where u1 and u2 are variables. Such a data structure is a skeleton representing all lists of length 2. This ability to embed variables in data structures and, perhaps, later constrain them leads to an elegant and powerful program style that is not immediately available to functional programmers.

Thus, it would seem, functional languages have something to gain from logic programming languages. However, conversely, it seems to us that the functional style also has several advantages over the logic style, for example, functional notation, typing, higher order capability, determinism and the existence of safe and efficient evaluation strategies in the sequential and parallel context. Our approach has therefore been to seek to **extend** the functional languages to achieve the capabilities outlined above without changing their nature fundamentally.

The way we have sought to do this is by introducing a programming structure into Hope termed **absolute set abstraction**. This construct is similar to the simple use of sets for iteration seen earlier (that was *relative* set abstraction) but much more powerful. Absolute set abstraction allows the programmer to specify sets of values by conditions, given as equalities between functional expressions, that the members of the set must satisfy.

For example given the append function defined normally

```
infix <> : 5
dec <> : list alpha # list alpha -> list alpha
--- nil <> ℓ ⇐ ℓ
--- (x :: ℓ1) <> ℓ2 ⇐ x :: (ℓ1 <> ℓ2)
```

using absolute set abstraction we could use it to split a given list, say [1, 2], using the expression

$$\{(\ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [1, 2]\}$$

which reads 'the set of all $\ell 1$, $\ell 2$ such that $\ell 1$ appended to $\ell 2$ equals [1, 2]'. This expression would evaluate to

```
{ ([], [1, 2]), ([1], [2]), ([1, 2], []) } : set(list num # list num)
```

Absolute set abstraction allows the introduction of logical variables into a functional language. The evaluation of such expressions entails an extension of the normal functional evaluation mechanism. The technique employed is known as **narrowing** but it is intuitively close to the resolution/unification process underlying logic programming. Thus for example in trying to find values for $\ell 1, \ell 2$ which satisfy the condition $\ell 1 <> \ell 2 = [1, 2]$ in the above expression we can make progress by unifying the condition against the equations of the program. Thus we can unify against the first equation for $<>$.

$$\{(\ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [1, 2]\}$$
$$\text{nil} <> \ell = \ell$$

and the substitutions

Input substitution	Output substitution
ℓ -> [1, 2]	$\ell 1$ -> nil
	$\ell 2$ -> [1, 2]

make the condition true. Thus, the expression is equivalent to

$$\{(\text{nil}, [1, 2])\}$$

There is another unification possible with the second equation for $<>$. Matching

$$\{(\ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [1, 2]\}$$

with

$$(x :: \ell 1) <> \ell 2 = x :: (\ell 1 <> \ell 2)$$

gives the substitutions

Input substitution	Output substitution
x -> 1	$\ell 1$ -> 1 :: $\ell 1$
$\ell 1 <> \ell 2$ -> [2]	$\ell 2$ -> $\ell 2$

The last input substitution is, of course, not a strict match. The way to read this is that the other substitutions make the two equations equal for any $\ell 1$, $\ell 2$ that satisfy the condition $\ell 1 <> \ell 2 = [2]$. Finding such values involves a recursive call and allows us to refine the above expression to

$$\{(1 :: \ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [2]\}$$

The two disjoint elaborations constitute independent contributions to the final answer and are to be unioned together; the exact analogue of or-parallelism in logic programming. Thus we have

$\{(\ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [1, 2]\}$

$\Rightarrow \{([], [1, 2])\} \cup \{(1 :: \ell 1, \ell 2) \mid \ell 1 <> \ell 2 = [2]\}$

Further, recursive, elaboration of the second component produces the result given earlier.

This extension to functional languages gives, we would claim, all the expressive power available in pure logic programming without doing violence to the rest of the language.

For example structures with embedded variables can be generated

$\{\ell \mid \text{length } \ell = 2\}$

$\{u :: (v :: nil)\}$: set(list alpha)

and all the programming styles that this allows in logic programming are now available to functional programmers.

As a final example we will recreate the traditional 'families database' example from logic programming in extended Hope.

```
data names == kate ++ heather ++ john ++ emma ++ bill ++
              annie ++ maurice

dec mum, dad : names -> names

--- mum heather <= kate

--- dad heather <= john

--- mum kate <= annie

--- dad kate <= maurice

--- mum john <= emma

--- dad john <= bill

dec parents : names -> set names

--- parents c <= {mum c, dad c}

dec grandparents : names -> set names

--- grandparents c <= {gp | p is_in parents c, gp is_in parents p}

grandparents heather

{maurice, annie, emma, bill} : set names

c st (kate is_in parents c)

heather: names

{ (f,p) | f heather = p } ! at the moment limited higher order unification is
                            allowed
```

```
{ (mum, kate), (dad, john), (parents, {kate, john}),
  (grandparents, {maurice, annie, bill, emma }) } : set (names -> alpha # alpha)
```

The extended Hope language is currently implemented by an interpreter, itself written in Hope, developed at Imperial College. As we shall see later in 5.1 we adopt a different approach to the compilation of these new constructs than conventional logic programming implementations and further work remains to be done on the integration of these features with the standard language and its implementation. We can report, however, that the extended language is proving very popular with its present user community.

4.2 Imposition of temporal ordering

As mentioned previously functional languages gain much of their power by allowing the programmer to abstract from notions of time. For many applications this lack of concern with the sequence of events occurring during the execution of the program is a boon to the programmer but there do exist application areas where concern with such orderings is central to the correctness of the solution. An example of such an application area is real time control, where the controlling program is not so much required to produce a value as organise a series of actions in the correct temporal sequence.

Our solution to the problem of applying functional languages to these areas is to stay with the basic pure functional language but to develop a separate language that allows a programmer to specify any temporal constraints that he requires imposed on the execution of his program. The program together with the temporal constraint is then automatically transformed to produce a single, augmented, functional program guaranteed to satisfy the temporal constraints under any evaluation regime.

Thus, for example, say a programmer had defined the following function to merge two lists

```
dec merge : list alpha # list alpha -> list alpha
    --- merge(nil, y) <= y
    --- merge(x, nil) <= x
m1: --- merge(c :: x, y) <= c :: merge(x, y)
m2: --- merge(x, c :: y) <= c :: merge(x, y)
```

Unconstrained evaluation could, correctly, produce any interweaving of the argument lists. Say, however, the programmer wanted the two lists to be merged alternately. Using our approach he would specify this requirement via a temporal statement referring to the **events** m1, m2 involved in the program's execution (an event is the application of a named equation). The temporal statement required would be

```
m1 -> Tomorrow(m2) and m2 -> Tomorrow(m1)
```

which, roughly, should be read 'if at any point in the execution a rewrite using m1 is performed then Tomorrow, i.e. at the next rewrite, a rewrite using m2 should be performed and vice versa'.

Our implemented transformation routine is able to take the above and produce the augmented merge program

```
dec merge : state # list alpha # list alpha -> list alpha
--- merge(s, nil, y) <= y
--- merge(s, x, nil) <= x
--- merge(ST(false, v2), c :: x, y) <= c :: merge(ST(true, false), x, y)
--- merge(ST(v1, false), x, c :: y) <= c :: merge(ST(false, true), x, y)
```

Thus any initial call to merge of the form merge(ST(false, false), $\ell 1$, $\ell 2$) would allow any of $\ell 1$, or $\ell 2$ to be passed on first but thereafter they would be merged alternately as required. Note that our method involves only the minimum necessary partial ordering being imposed on the events rather than the total ordering imposed when an imperative language is used.

Observant readers will have noted that our equations for merge have overlapping left hand sides and therefore specify a non-deterministic system and not a proper function. Indeed they do and there are many ramifications to our approach that we do not have space to go into here but we feel that it does represent a route whereby the functional style can be applied naturally to areas such as real time control, transaction processing and operating systems while still retaining the pure declarative spirit.

5 Program transformation

Program transformation is the process of converting a program into an alternative form, one that is semantically equivalent to the original, i.e. it computes the same function, but differs in some quantitative aspect, e.g. execution time or space utilisation. As developed, program transformation aims to be a **constructive** process, as much as possible we look for manipulation rules or operations that can be applied to programs systematically to produce new versions which will be guaranteed equivalent to the original rather than inventing new versions and then **verifying** that they are equivalent to the original.

Transformation also aims to make order of magnitude improvements in the efficiency of programs. Thus, in contrast to conventional compiler optimisation, it attempts to formalise and systematise the process of **program design** or **algorithm invention**. Thus the starting point could be some very high level specification of the problem to be solved and the end result an efficient program to accomplish the specified task. Transformations are normally source to source, i.e. the original and transformed program are in the same language, in our case extended Hope. Transformation has been applied to

imperative languages but, in our opinion, the greater mathematical tractability of the declarative languages affords much better prospects for successful application of transformation techniques.

Transformation operations can be divided into two types: the automatic and the non-automatic. For operations in the former category static analysis of the program can reveal safe opportunities for transformation, whereas in the latter category the desired transformations have to be accomplished via the composition of smaller guaranteed safe operations and there are no automatic procedures guaranteed to produce the required composition. In this case theorem proving or user guidance is required, but as we shall see later the process is often sufficiently systematic or intuitive for this not to be too daunting a prospect. As it is the intellectual process of program design that we are attempting to formalise it should be clear that full automation will only be realised with the advent of genuine artificial intelligence capabilities.

We will give brief examples of both styles of transformation.

5.1 Non-decidable transformations: removal of unification

Definitions employing absolute set abstraction tend to be succinct and expressive, as we saw above, however, this expressive power is often at the cost of inefficient execution. These inefficiencies arise for two reasons: firstly the abstract machine operations required to support absolute set-abstraction, unification and exploration of an AND-OR search space are more complex than those required for standard functional evaluation; and secondly the expressive power of absolute set abstractions encourages a style of description that is inherently inefficient. We do not regard this as a disadvantage but seek to get the best of both worlds by encouraging free use of absolute set abstractions and then using transformation to convert these definitions to equivalent definitions employing only standard functional apparatus.

Traditionally, formally based transformation systems, such as the unfold/fold system[2], have been presented as a set of rules allowing systematic manipulation of program equations. The close affinity between many of these rules and the manipulations carried out during actual program execution has long been observed and the term **symbolic evaluation** coined to describe the 'execution' of programs on symbolic rather than actual data. With the introduction of unification into the language which allows the run time support of logical variables this affinity between symbolic and actual execution becomes an identity, resulting in a pleasing simplification.

For example, using absolute set abstraction we can define a function *front* that returns the initial segment of a list of given length by

```
dec front : num # list alpha -> list alpha
--- front(n, ℓ) ⇐ ℓ1 st (ℓ1 <> ℓ2 = ℓ, length ℓ1 = n)
```

The st (read 'such that') construct is used when we know that the set defined by the conditions is a singleton.

In 2, we saw how narrowing would allow us to evaluate the right hand side of front with ℓ and n bound to some value (ground term) producing instantiations for the variable $\ell 1$. There is nothing to stop us, however, from executing front with all the variables non-ground.

Matching the first condition of

$$\text{--- front}(n, \ell) \Leftarrow \ell 1 \text{ st } (\ell 1 <> \ell 2 = \ell, \text{length } \ell 1 = n)$$

with

$$\text{nil} <> \ell = \ell$$

using exactly the same process as before allows us to rewrite it to

$$\text{--- front}(n, \ell) \Leftarrow \text{nil st length nil} = n$$

Matching the remaining condition with

$$\text{length nil} = 0$$

instantiates n to 0. Note that in this case we are instantiating a bound variable so the new equation produced is

$$\text{--- front}(0, \ell) \Leftarrow \text{nil}$$

Returning to the starting definition of front

$$\text{--- front}(n, \ell) \Leftarrow \ell 1 \text{ st } (\ell 1 <> \ell 2 = \ell, \text{length } \ell 1 = n)$$

and matching with

$$(x :: \ell 1) <> \ell 2 = x :: (\ell 1 <> \ell 2)$$

gives us

$$\text{--- front}(n, x :: \ell) \Leftarrow x :: \ell 1 \text{ st } (\ell 1 <> \ell 2 = \ell, \text{length}(x :: \ell 1) = n)$$

matching with

$$\text{length}(x :: \ell) = 1 + \text{length } \ell$$

gives us

$$\text{--- front}(n + 1, x :: \ell) \Leftarrow x :: \ell 1 \text{ st } (\ell 1 <> \ell 2 = 1, \text{length } \ell 1 = n)$$

The conditions on the right hand side of the above equation are now

identical with the conditions of the original definition. So rather than continue the execution we can replace them with a call to front getting

--- front(n + 1, x :: ℓ) $\Leftarrow$ x :: ℓ1 st ℓ1 = front(n, ℓ)

or more succinctly

--- front(n + 1, x :: ℓ) $\Leftarrow$ x :: front(n, ℓ)

Thus by simple evaluation and recognition of a recurrence we have produced two new equations for front

--- front(0, ℓ) $\Leftarrow$ nil
--- front(n + 1, x :: ℓ) $\Leftarrow$ x :: front(n, ℓ)

which are in the form required and constitute a much more efficient program for front than the previous definition.

5.2 An automatic transformation: memoisation

Memoisation is a simple, appealing, program improvement operation that was suggested some while ago[3] but only recently have developments[4,5,6] made it practical.

The idea behind memoisation is that programs are often inefficient because they repeat computations unnecessarily. In functional languages this can arise when a function is applied to an argument equal to one it has been applied to previously. Because of the lack of side-effects any function is guaranteed always to return the same value when applied to the same argument, so why not remember the value and the result in a table and simply look it up if it is subsequently needed?

The trick of course is knowing when and what to remember and, less obviously, knowing when to discard entries from the table. Work at Imperial[6], part of the Flagship research programme, solves these problems. A set of theorems have been proven that allow static analysis of Hope programs not only to establish when memoisation is possible and beneficial, but also to automatically produce a method that will maintain the memo table in the optimal manner. The maintenance of the memo-table is performed by a function, called the **table manager** and syntherises for each function to be memoised, which when applied to a value about to be stored in the table returns the values whose entries can now be deleted as the logic of the algorithm implies they will never be required again. These methods have been implemented and can automatically synthesise the appropriate table manager functions.

Memoisation can produce startling results. Consider the traditional Fibonacci function

```
dec fib : num -> num
--- fib 0 <= 1
--- fib 1 <= 1
--- fib (n+2) <= fib(n+1) + fib n
```

This function is beloved of transformationists because it exhibits exponential redundancy. Applying our memoisation technique produces the table manager function, TBM say,

```
dec TBM : num -> num
--- TBM n <= n-2
```

Thus if we trace an application of a version of fib, augmented with a memo-table represented as a list of pairs, initially empty, to 5 we get

Expression being evaluated	Memo Table
fib 5	[]
⇓	
•	
•	
•	
⇓	
((fib 2 + fib 1) + fib 2) + fib 3	[]
⇓	
•	Recursive evaluation provides memo-table entries for fib 3 and fib 2
•	
•	
⇓	
(3 + fib 2) + fib 3	[(3, 3), (2, 2)]
⇓	fib 2 is evaluated by table look-up rather than recursive evaluation
(3 + 2) + fib 3	[(3, 3), (2, 2)]
⇓	The completion of the evaluation for fib 4 results in its entry in the table **and** the deletion of the entry for TBM(4) i.e. 2
5 + fib 3	[(4, 5), (3, 3)]
⇓	fib 3 is evaluated by table look-up
5 + 3	[(4, 5), (3, 3)]
⇓	The value for fib 5 is added to the table and the entry for TBM(5) i.e. 3 is deleted
8	[(5, 8), (4, 5)]

Thus the memo-table for fib never grows larger than two entries and the exponential algorithm has been converted to a linear one with dramatic improvements in efficiency.

5.3 Range of transformation operations

The transformation work aims to provide a **complete** program development methodology. That is any correct implementation of a specification should be achievable via some transformation sequence. Furthermore it is desirable that transformation corresponds to some intuitive program design process. To meet these ends transformation techniques have been applied to the following areas amongst others

(i) non-automatic, partial evaluation

- loop combination
- introduction of parallelism
- removal of unification
- abstract data type implementation
- conversion to iterative form

(ii) automatic

- memoisation
- imposition of temporal constraints
- synthesis of function inverses
- strictness analysis
- structure overwriting
- mode analysis
- linearisation

In the non-automatic cases the underlying methodology for each application is the same (unfold/fold or partial evaluation) but the domain of application provides heuristic guidance for the process and, as we will see later, gives structure to the transformation.

5.4 Transformation based programming environments

The goal of the transformation work within Flagship is the development of a prototype program development and maintenance system where the underlying program development methodology is formally-based transformation.

Most transformation work has involved the development of systems to assist the transformation process. The question is what sort of system should we aim for? As we stated earlier full automation is a long way off. The other extreme, where all the guidance is provided by the user and the system simply checks the applicability of the indicated transformation and performs the appropriate book-keeping, is easy to implement and certainly represents an advance on purely pen and paper use of transformation but would prove too tedious and painstaking for everyday use.

The style of system we, in Flagship, are aiming for is one where the user provides the strategic guidance but is able to convey his intentions to the system in a high level and structured manner and the system is able to take over and implement these plans by producing, from them, a complete transformation to achieve the desired effect.

The route we are adopting to such a system uses the ideas of **meta-programming** pioneered by the ML-LCF project[7]. A programmer designing a transformation concentrates his attention on developing a meta-program which when applied to the object program (the specification) produces the desired efficient version.

We are, not surprisingly, using extended Hope for our meta-programming language. The operations, functions, defined in the meta-language correspond to transformation operations. The lowest level functions correspond to the lowest level, meaning-preserving, transformation operations, i.e. decidable operations such as memoise or storage-overwrite, or single partial evaluation steps. Out of these, higher level transformation operations can be created using the normal function definition apparatus. The beauty is that because the defined operations inherit the meaning-preserving nature of the base operations there is no way the meta-programmer can produce an incorrect transformation. A meta-program may fail, indicating that the intended transformation is inappropriate, but if it succeeds the produced program is guaranteed equivalent to the input program.

For example we can define meta-functions *remove_unification* and *memoise*

```
dec remove_unification : hope_program # additional_information ->
                                                    hope_program

dec memoise : hope_program # additional_information -> hope_
                                                        program
```

that when applied to our previous examples would produce the results obtained earlier.

The meta-programmer can build further on these to produce specific structured meta-programs for a particular transformation or more powerful, generally applicable, transformation tactics.

Thus the programmer's development activity is focussed on creating the system specification and the appropriate meta-program. When this has been successfully accomplished he has a concrete, structured and executable representation of the complete system design which can be stored, queried and modified.

We need to ensure that the system designer has sufficient expressive power via the meta-language to achieve any implementation he can conceive even at the cost of some tedium. This is achieved by the general-purpose nature of

the lowest level transformations. Having established this bottom line we are then at liberty to attempt to give the (meta) programmer as much expert assistance from the system as possible. For example, tactics such as the remove_unification tactic can exist in two forms. One, where enough information is given via the second parameter to ensure a successful transformation without any searching; and another where no help is given and the system has to engage in some theorem-proving or searching activity to attempt the desired transformation, producing, if successful, the information that will make the re-play automatic.

Thus the system will work in an exploratory manner assisting the designer in producing the meta-programs and in a direct manner implementing a fully defined transformation plan.

The same ideas of meta-programming can be applied to formalise the non-meaning preserving transformations used in system development, i.e. the operations of specification creation and modification. Thus, for example, we would like to express module linking and program modification or enhancement not as textual operations on the object program but as application of meaningful meta-language operations. Thus all aspects of system development could be recorded in a formal concrete object, the meta-program.

6 Conclusions

Functional programming technology offers many advantages to the computer industry. One task of Flagship is to make these promises actualities.

References

1 BURSTALL, R.M., MACQUEEN, D.B. and SANNELLA, D.T.: 'HOPE: An experimental applicative language'. Proc. 1980 LISP conference, Stanford, California, 136–143.

2 BURSTALL, R.M. and DARLINGTON, J.: 'A transformation system for developing recursive programs'. JACM 1977, 24, 1, 44–67.

3 MICHIE, D.: 'Memo functions and machine learning'. Nature 1968, No. 218, 19–22.

4 HUGHES, J.: 'Lazy memo-functions'. Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 1985, 129–145.

5 KELLER, R.M. and SLEEP, M.R.: 'Applicative caching: Programmer control of object sharing and lifetime in distributed implementations of applicative languages'. Proc. ACM Conference on Functional Languages and Computer Architecture, Wentworth, Mass., 1981.

6 KHOSHNEVISAN, H.K.: 'Memoisation as a practical alternative to program transformation'. Internal report, Functional Programming Section, Department of Computing, Imperial College.

7 GORDON, M.J., MILNER, A.R.J. and WADSWORTH, C.: 'Edinburgh LCF'. Report CSR-11-77, 1977, Dept. of Computer Science, University of Edinburgh.

Dactl: A computational model and compiler target language based on graph reduction

J.R.W. Glauert, J.R. Kennaway, M.R. Sleep
Declarative Systems Project, University of East Anglia, Norwich NR4 7TJ, England

Abstract

The Alvey programme has adopted a model of computation based on Graph Reduction as a focus for UK work on New Generation Languages and Architectures. The development of this model is being undertaken by an Alvey-sponsored project at the University of East Anglia (UEA) at Norwich entitled Dactl (Declarative Alvey Compiler Target Language). Collaborative partners in the Alvey project are ICL, Imperial College, and Manchester University.

The motivations and design goals for Dactl are presented. Both Language and Architecture design influences are discussed. Dactl0 – the current release of Dactl – is described.

The major changes envisaged in Dactl1 are discussed. The design of Dactl1 has been heavily influenced by the collaboration between the authors and the Dutch Parallel Reduction Machine project.

1 Introduction: motivations and design goals for Dactl

1.1 Motivations

The main motivation for work on the Declarative Alvey Compiler Target Language (Dactl) is to develop a computational model which can act as a bridge between the designers of new generation languages (e.g. Hope, ML, Parlog) and appropriate new generation parallel architectures (e.g. Flagship, GRIP, ZAPP). Such a bridging model of computation provides several major benefits:

(a) It acts as a focus for language and architecture developments at a time when both are changing fast.
(b) It offers an increasingly secure interface between many languages and many parallel architectures, giving a strong entry card to discussions on international collaboration.

(c) It provides new insights into parallelism via theoretical and simulation studies.

Because declarative languages and architectures are evolving so rapidly, and attempts to stabilise at either the language or architecture level appear premature, the above motivations are particularly strong.

1.2 Design goals

These and other considerations led the Alvey directorate to set up a design team for Dactl in July 1984. The following requirements were identified:

(a) Support for languages with a strong declarative flavour, e.g. Hope, LISP, Prolog.
(b) Wide scope for exploring novel hardware designs.
(c) The ability to express details of evaluation order, and whether a result is shared or copied. However, Dactl explicitly abstracts from lower level details such as processor scheduling or memory management.
(d) An underlying formalism which supports both human and mechanical efforts to reason about both the results and the performance of programs.
(e) The ability to interface to existing software and hardware, and take advantage of special purpose processors.

1.3 Releases of Dactl

Three versions of Dactl were planned, Dactl0, Dactl1 and Dactl2. Each represents a stage in our increasing understanding of the key issues, both practical and theoretical.

Dactl0 has been on limited release since 1985 to elicit feedback. It represents a guess at subsequent and future developments in the Declarative Systems area.

Dactl1 is intended to involve a significant revision of Dactl0 which takes account of available feedback, and also the active collaboration which has taken place between the UEA design team and the Dutch Parallel Reduction Machine Project.

Dactl2 will be based on the experience gained with Dactl1, and take account of the possibility of harmonising with similar work elsewhere with international collaboration in mind.

1.4 Organisation of the paper

Section 2 describes some technical background to the development of Dactl. Section 3 gives a detailed overview of Dactl0. Section 4 discusses the major changes envisaged in Dactl1. The annexe to the paper is an edited version of the original Dactl0 report[7].

2 Technical background

A computational model must be both *thinkable* and *do-able*. This means we must consider both language and architecture influences if we are to devise a successful model. However, there were no clear winners on either the language or architecture front when Dactl work was started. Consequently, the Dactl design team identified key features from the best available work in declarative languages and declarative architectures. The following sections record the principal influences.

2.1 Language considerations

Declarative languages have dual readings. On the one hand a program can be read as a set of constraints which determine *what* is computed by the program. On the other hand, a program can be read as a set of rewrite rules which govern *how* the computation is performed.

A declarative programmer needs to keep in mind both readings to produce programs which are both correct and efficient. A functional programmer will rely on the lambda calculus for his declarative reading, and on some reduction model of computation for his operational reading. A logic programmer will use the predicate calculus for his declarative reading, and some theorem-proving model of computation, such as resolution, for his operational reading.

Interestingly, the operational readings for both logic and functional programs are both based on viewing the clauses or equations as rewrite rules. Work by Reddy[14] and others suggests there may be an underlying model of computation which supports both styles of declarative programming.

Like Reddy, we are interested in computational models which bring together previously distinct worlds. Unlike Reddy, we have taken the expression of control information in such a model as being of paramount importance. We start with a Term Rewrite model of computation[10] and generalise it in a number of ways:

(a) In the Dactl model, rewrites take place on graphs rather than trees.
(b) These rewrites may be done in parallel.
(c) Explicit control over evaluation order can be expressed.
(d) Explicit control over whether rewrites are performed using copying or sharing techniques, or both, can be expressed.

2.2 Architecture considerations

In the most successful model of computation to date, that proposed by von Neumann, a single processing element repeatedly shuffles a single word of information between a set of local registers and a large global memory with

serialised access. Each instruction, except the last, appoints exactly one successor instruction.

New Generation architectures will generalise the von Neumann model in a number of ways of which the following are particularly notable:

(a) Concurrent access to memory by many processing elements.
(b) Instructions may appoint an arbitrary number of successors.

In principle we may generalise the von Neumann memory to support parallel access by attaching to each word its physical address. This allows us to regard the memory as an unordered set (or *pool*) of *packets*, each of which contains an explicit representation of both the address (name) of a packet, and its contents (which may of course include the addresses of other packets). There will be one packet for each word in the corresponding von Neumann memory, and *all operations on the pool of packets must maintain this property.*

Having thus decomposed the von Neumann memory, it is possible to envisage a large number of processing elements (PEs) operating concurrently on the set of packets. Each PE has the right to temporarily remove any available packet from the pool, and return it with possibly modified contents. During the period of removal, the packet is unavailable, and PEs attempting to access it are *blocked.* Different packets can be operated on simultaneously. Note that a PE can never change the address field of a packet in this model. Parallel architectures which adopt this view are called *Packet Communication Architectures* (PCA).

The Manchester Dataflow Machine[18] is an early example of how hardware might support such concepts efficiently. The recently commissioned ALICE machine[4] supports a more general notion of Packet Communication. The influence of the Packet Communication model on the design of Dactl0 can be seen in the operational description of Dactl0 semantics in the Annexe.

2.3 Level of the Dactl interface

The level of Dactl is intended to lie somewhere between the murky world of low-level resource scheduling and much more abstract worlds such as the lambda calculus or mathematical logic. A compiler which produces Dactl is intended to employ sophisticated transformations aimed at, for example, minimising the number of Dactl rewrites during a computation. Similarly, the code generator which translates Dactl to code for a particular machine may employ advanced optimisation techniques.

Note that there is an underlying assumption that Dactl will provide *metrics* for guiding code generators. A suitable evolution of Dactl should ensure that the compiler writer loses very little in efficiency if he aims at code which will minimise some application dependent function of the space/rewrite metrics of Dactl. Conversely, a machine designer should work towards minimising the real time/space requirements of some synthetic Dactl computation.

3 Introduction to Dactl0

In this section we outline the main features of Dactl0. Readers requiring more detail are referred to the annexe, which contains the syntax and operational semantics of Dactl0 as originally circulated to a limited audience in the 1985 report[7]. The annexe also contains many examples of Dactl0 programs.

3.1 Rule based computation

Dactl0 is a notation for describing how sets of rules may operate in parallel on some expression to compute a result. Rule-based programming will be familiar to those working with functional and logic languages. For example, the equations:

$$\text{fib}\ (0) = 1$$
$$\text{fib}\ (1) = 1$$
$$\text{fib}\ (n) = \text{fib}(n-1) + \text{fib}(n-2)$$

specify the well-known fibonacci function using a *priority* rule system[1] in which earlier rules take precedence over later rules in case of conflict. Note that some such notion of priority must be present, since otherwise the expression fib(0) would match both the first and the third of these rules, and the intention is that only the first rule should apply.

Most implementations of functional languages are based on using the equations as rewrite rules which (at least notionally) repeatedly operate on matching components of the expression until no more rules apply, and any such resulting form (a *normal* form) is the result of the computation. Apart from disambiguating meta-rules (e.g. textual priority as illustrated above, and Dactl0's *specificity* described in section 2.4, annexe), a rule-system does not specify any order of rewriting co-existing *redexes*. Nor does it specify the representation of expressions: in principle either string or graph reduction may be employed. A major property of Dactl0 is that such decisions can be made explicit.

3.2 Dactl0 terms

Two styles for defining terms appear in the literature[11]. In the *functional* style, each term has two attributes: a function symbol, and an (ordered) list of subterms. In the *applicative* style, there is a set of function symbols of arity 0, and a single functional symbol of arity 2, called application, and denoted by juxtaposition. For example, the term representing a list consisting of head h and tail t would be represented as cons(h,t) in the functional style, and (cons h t) in the applicative style.

Terms written in the applicative style can be translated into the functional style simply by making the implicit binary application operator explicit. The

functional representation of an applicative term such as (cons h t) is Ap(Ap(cons,h),t). But the distinction between the two styles is not merely a matter of taste, as the translation does not go the other way. For example, the applicative term (cons h t) has as a subterm the applicative term (cons h), whose functional representation is Ap(cons,h). But there is no corresponding subterm of the functional term cons(h,t).

Dactl0 lies somewhere between the applicative and functional styles, and we call it *polyapplicative*. Its representation of terms is based on *tuples* of arbitrary size. A Dactl0 term is either an *atom* (which we may think of as a nullary function symbol), a *variable*, or a *tuple* of terms. Considered functionally, there is an infinite number of tuple constructors, one for each size of tuple. Textually, tuple terms are denoted by grouping together immediate subterms with angle brackets. Thus the Dactl0 term <cons 1 nil> is a 3-tuple, whose immediate subterms are the atoms cons, 1 and nil. In the functional style, the representation of this term might be Tuple3(cons,1,nil). Note that the first position in a tuple is *not* special in Dactl0.

3.2.1 Expressing sharing: To describe Dactl0 terms which involve sharing or cycles, identifiers and **where** definitions are used. Identifiers are textually distinguished from atoms by beginning with the symbol "$". Thus the (graph) term:

```
<cons $x $x> where $x:<cons 1 nil>
```

denotes a graph, the root of which is a triple whose first component is the atom "cons", and whose second and third components are references to another node, a triple <cons 1 nil>. There is only one copy of that second node in the graph, which both the occurrences of $x refer to. Such a graph can be used to represent the (tree) term <cons <cons 1 nil> <cons 1 nil>>, sharing the occurrences of the repeated subexpression.

Cyclic graphs may be written. The term $x:<cons 1 $x> specifies a cyclic graph which might be the representation of an infinite list of 1's, in the translation of some language to Dactl0. Note that Dactl0 itself is simply a language for expressing parallel graph manipulation, and has no such interpretation built in.

Conditions under which a graph representation of term rewriting is both sound and complete are given in[2].

3.2.2 Matching: Dactl0 patterns are required to contain at most one occurrence of any variable (this is the *left-linearity* condition). Thus all patterns in Dactl0 are trees. A matching of a Dactl0 pattern to a Dactl0 program graph consists of a structure-preserving many-one mapping from the pattern to the program graph. The subgraph identified by the matching is called the *contractand.*

A *(Dactl0) redex* consists of a contractand together with an arc pointing to its root. Unlike a redex in Term Rewriting, a Dactl0 redex must specify a particular in-arc (called the *firing* arc in what follows).

3.3 Expressing control flow

Arcs in a Dactl0 graph may be marked with a ! to indicate a *firing*. Syntactically, this is done simply by allowing all term references to be preceded by a !. Semantically, it is important to note that such a firing is in fact attached to an *arc* in the graph and *not* the root node of a term. Informally, a firing is a request to the evaluator to attempt a single rewrite at the root node of the term indicated, and report the result back to the parent tuple containing the fired arc.

As an example, the term !<+ !<* 3 4> !<* 5 6>> specifies all the firings necessary to evaluate the expression 3*4+5*6 to the integer result 42, assuming that suitable rules for the atoms + and * have been defined.

Each firing corresponds to a bounded amount of work for the architecture. This provides a basis for reasoning about performance. But it also means that careful attention must be paid to markings if the expected normal form is to be obtained. In Dactl0, *a term with no firings is in normal form.*

Control is expressed in two ways in Dactl0. Firstly, the arcs in the graph to be rewritten may be marked with (initial) firings. Secondly, terms specified on the right hand sides of rules may contain zero or more firings.

3.4 Dactl0 model of computation

The Dactl0 model of computation uses ideas from three previously distinct schools, namely reduction, dataflow, and Petri net models of computation.

- As with Petri net models, the state of a computation is described by markings (such as !) on the arcs of a graph.
- As with reduction models, a fired node (that is, one which has a firing on one or more input arcs) is matched against a set of rules, and a matching rule is executed to modify the graph.
- As with demand driven dataflow models, a firing leads to a (reference to a) result flowing back up the firing arc into a 'hole' in the node responsible for the original firing. The Dactl terminology for this last process is *redirection*, since another way of describing the operation is to say that the fired arc is redirected to point to the result.

3.4.1 Rewriting in Dactl0: As can be seen by examining the examples in the annexe, apart from syntactic details Dactl0 rules look very much like the rules encountered in the equational style of programming[13]. They usually have just one right hand side. However, to express concepts such as sharing

versus copying, and state dependent computations, the general form of a rule is as follows:

```
LHS := OVERWRITE ⇒ REDIRECTION
```

The general form of a Dactl0 rule allows two right-hand sides, one which specifies how the root node of the contractand changes, and one which specifies the subgraph to which the firing parent is redirected. If only the overwrite part is given, no redirection at all takes place. If only the redirection part of a rule is given, rules which copy rather than share can be written.

As an example, consider the function from(n) which is frequently used in functional programs to specify the infinite list of integers (n,n+1,n+2,.....). The Dactl0 rule:

```
<from $n> := <cons $n <from !<+ $n 1>>>
```

expresses a sharing version of the from function, in which evaluation causes a change of form which benefits other users. On the other hand, such sharing may force an implementation to maintain references to arbitrarily large terms, and it may be preferable to recompute to save space. The Dactl0 rule:

```
<from $n> ⇒ <cons $n <from !<+ $n 1>>>
```

omits the overwrite step, creates a new subgraph, and redirects the firing parent to it; only the firing parent experiences the benefit of the evaluation.

We now explain how rules which use both the overwrite and redirection options work. First recall that a redex in Dactl0 consists of a subgraph (the contractand) and a fired arc pointing to its root. The two right hand sides in the general form of a rule say what the rewrite does to the contractand and the fired arc respectively. The OVERWRITE section specifies a term to replace the contractand. This replacement is done by overwriting the root node of the contractand by the root node of the OVERWRITE term. Thus – with the possible exception of the firing arc – the effect of the rewrite is shared. The REDIRECTION section specifies a term to which the firing arc alone is redirected, possibly with a firing. Operationally, the redirection is performed by overwriting the relevant field in the tuple which has as an immediate subterm the fired arc. If the REDIRECTION section is missing, the firing arc remains, picking up any firing associated with the OVERWRITE term. If the OVERWRITE section is missing, redirection alone takes place.

EXAMPLE: Use of rewriting and redirection to implement a selector function.

The well-known law defining the head of a list can be implemented in Dactl0 as:

```
<head <cons $h $t>> := <ind $h> ⇒ $h
```

where ind is user defined (e.g. by the rule <ind $i> ⇒ $i).

To take a specific instance of the rule for head, which is illustrated in Fig. 1, suppose we have a Dactl0 graph containing a part which looks like:

```
$parent:<+ 1 !$sub:<head <cons $hd:<+ 2 3> $tl:nil>>>
```

where the names $parent and $sub are introduced for reference, identify the root of the whole graph, and the root of the matching subgraph respectively. Informally, both are pointers, as are $hd and $tl. The matching process identifies a contractand consisting of the fired arc which is the second component of the tuple identified by $parent, and the subgraph identified by $sub. The matching also identifies the binding $h = $hd, and $t = $tl.

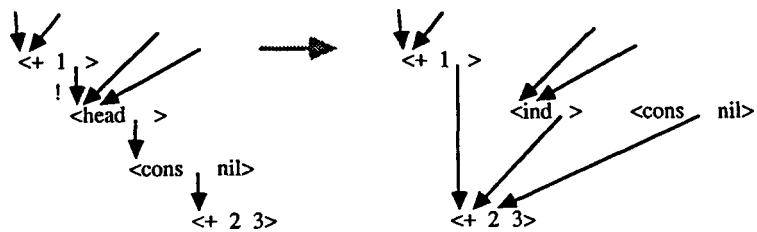


Fig. 1

Having established the redex, the Dactl0 rewrite now proceeds. First, two new graphs are constructed according to the REWRITE and REDIRECT portions respectively. Appropriate bindings replace variables.

Next, the root of the REWRITE term so constructed overwrites the root of the redex. In this case, the effect of the REWRITE step is to cause <head $> to be overwritten with <ind $> **where** $:<+ 2 3>.

Finally, the redirection is performed by replacing the fired arc by a reference to the new REDIRECT term. Redirection is used both to reactivate the parent and also to return a result associated with the firing. It is the basis for synchronisation in Dactl0.

Considerable control may be exercised over firing order if desired. This is done by marking the arcs on the right-hand side of a rule with a !, representing a firing. Strict operators, such as +, require both their arguments to be evaluated, suggesting parallel firings. For example, the rule:

```
<parsum $e1 $e2> := !<+ !$e1 !$e2>
```

will fire $e1 and $e2 concurrently. When their evaluation terminates, the rule for + (which expects both its arguments to be evaluated) will come into play.

The ability to fire more than one subterm removes the von Neumann constraint which requires each instruction to appoint a unique successor.

3.5 Examples

We now consider some familiar examples and the various forms they might take in Dactl0.

3.5.1 Sequential summation of a tree of integers: We begin with a Dactl0 program which sums the leaves of a tree of integers, enforcing sequential execution:

```
program sumtree;
imports arithmetic;
atom sum; sum1; pair; leaf;
rule
<sum <pair $h $t>> := !<sum1 !<sum $h> <sum $t>>;
<sum <leaf $t>> => !$t;
<sum $t> := !<sum !$t>;
<sum1 $Inum $t> := !<+ $Inum !$t>;
rewrite
!<sum <pair <pair <leaf 2> <leaf 3>> <pair <leaf 4> <leaf 5>>>>;
endprogram sumtree;
```

The **imports** statement imports from outside the program atoms and rules which implement integers and some standard arithmetic operations. Such a module looks to the program like another piece of Dactl0 code, though it may be implemented (as will almost certainly be the case for integer arithmetic) by some hardware device. We anticipate that it will be possible to make much existing software (and for that matter special purpose hardware) look like Dactl0 modules.

In the graph to be rewritten in this example only the root node has been fired. (More precisely, the firing is placed on a notional arc leading from the evaluator to the root node.) This is because we want to describe the usual sequential control flow, starting at the root. Any marking of the initial graph is allowed in Dactl0.

When this program is run, a graph is constructed corresponding to the term (expression) specified in the rewrite part. The general idea is that the machine examines all fired nodes in the graph, to see if the terms they represent can be matched against the left-hand sides of the rules specified in the rule section of the program. Where a match is detected, the changes specified by the rule which matches are made. A fundamental property of Dactl0 is that only the root node of a matching term is overwritten, and the action is atomic.

As the computation proceeds, activity may spread to other nodes in the graph. In the example run shown below we indicate with a # those nodes whose further firing is held up awaiting replies from children. Such nodes are reconsidered for matching when replies are received. The state transitions

involved in the example are as follows. We have omitted trivial events connected with firing of integers and + operations.

```
!<sum <pair <pair <leaf 2> <leaf 3>> <pair <leaf 4> <leaf 5>>>>
:= #<sum1 !<sum <pair <leaf 2> <leaf 3>>> <sum <pair <leaf 4> <leaf 5>>>>
:= #<sum1 #<sum1 !<sum <leaf 2>> <sum <leaf 3>>> <sum <pair <leaf 4> <leaf 5>>>>
:= #<sum1 !<sum1 2 <sum <leaf 3>>> <sum <pair <leaf 4> <leaf 5>>>>
:= #<sum1 #<+ 2 !<sum <leaf 3>>> <sum <pair <leaf 4> <leaf 5>>>>
:= #<sum1 !<+ 2 3> <sum <pair <leaf 4> <leaf 5>>>>
:= !<sum1 5 <sum <pair <leaf 4> <leaf 5>>>>
:= #<+ 5 !<sum <pair <leaf 4> <leaf 5>>>>
:= #<+ 5 #<sum1 !<sum <leaf 4>> <sum <leaf 5>>>>
:= #<+ 5 !<sum1 4 <sum <leaf 5>>>>
:= #<+ 5 #<+ 4 !<sum <leaf 5>>>>
:= #<+ 5 !<+ 4 5>>
:= !<+ 5 9>
⇒ 14
```

3.5.2 Parallel summation of a tree of integers: Although our first example leads to purely sequential computation, this is entirely a property of the particular rule system. Dactl0 allows us to write rules which cause parallel firings. We can take advantage of this to write another version of the above example which allows a suitable architecture to exploit parallelism:

```
program parsumtree;
imports arithmetic;
atom sum; pair; leaf;
rule
 <sum <pair $h $t>> := !<+ !<sum $h> !<sum $t>>;
 <sum <leaf $t>> ⇒ !$t;
 <sum $t> := !<sum !$t>;
rewrite
 !<sum <pair <pair <leaf 2> <leaf 3>> <pair <leaf 4> <leaf 5>>>>;
endprogram parsumtree;
```

Each time the first rule for sum is activated, two child arcs are fired (nominated), and a suitable architecture could execute these firings concurrently. Note that the parallel version has fewer rules than the sequential ones. This is because the parallel version delegates decisions regarding evaluation order to the architecture, and therefore does not need to specify them.

The relevant state transitions are shown below, again using the # to indicate nodes whose firing is suspended awaiting replies (as redirections) from their children.

```
!<sum <pair <pair <leaf 2> <leaf 3>> <pair <leaf 4> <leaf 5>>>>
:= #<+ !<sum <pair <leaf 2> <leaf 3>>> !<sum <pair <leaf 4> <leaf 5>>>>
:= #<+ #<+ !<sum <leaf 2>> !<sum <leaf 3>>> #<+ !<sum <leaf 4>> !<sum <leaf 5>>>>
= #<+ !<+ 2 3> !<+ 4 5>>
:= !<+ 5 9>
⇒ 14
```

This example can be seen as constructing a dataflow tree from the top down, and integers flowing up the tree from the leaves to be combined by + nodes.

The first two rules assume that firing an arc will result in either a reference to an integer or a pair of integer trees. The third rule for sum deals with the case where neither rule applies. It simply fires its argument, assuming that this will yield a result with which the earlier rules can deal. This allows quite general expressions to be accommodated, providing they guarantee to return either an integer leaf node or a pair of integer trees. Generalisation of such reasoning may lead to useful proof rules for Dactl0.

3.5.3 A simple unique identifier server: It is possible to write state dependent rules in Dactl0, for example a simple *unique identifier* server which changes its state on each firing:

```
<newname $n> := <newname !<+ $n 1>> => $n
```

The firing parent sees $n as the reply. Other references to the newname node see an incremented version of the newname. Note that if many firings exist on a given node, the Dactl0 virtual machine will select only one at a time, so atomicity is achieved.

The presence of state makes it more difficult to reason about compositions of such rules. In Hewitt's terms they describe *impure* actors[9]. However, there are situations where they may represent just what the implementor intends to express. For example, the newname rule expresses the notion of a counting register which increments on each firing, and responds with the old value. Every firing client will receive a distinct integer as a reply.

3.5.4 Synchronisation: The usual rule for Dactl0 matching is that the matching process is delayed when it encounters a fired arc. For example, given the term:

```
!<sync f !<computation1 ....> !<computation2 ...>>
```

and the rule:

```
<sync $f $c1 $c2> := !<$f $c1 $c2>
```

replies must be received (i.e. redirections must occur) for both computation1 and computation2 before the sync rule will fire, invoking f with the results of the two computations as arguments.

Note however that the meaning of receiving a reply is entirely in the hands of the generator of Dactl0 code. At one extreme, receiving a reply could mean that the result is in some ground form. At the other extreme, it could mean that just one rewriting has taken place. In Dactl0 the meaning of a reply is entirely a property of the rule system and its markings.

Certain computations – for example bottom-avoiding merge – require a non-deterministic form of synchronisation which operates even when one of the

computations is still in progress. Dactl0 supports such concepts by allowing firings to be associated with the anonymous variable on the left hand side of a rule. Thus !$ may be used in a pattern in Dactl0. It is the only pattern which matches a fired arc. Its anonymous nature avoids the problem of defining the meaning of a firing in progress.

This completes the introductory section. A more complete description of Dactl0, together with further examples, occurs in the annexe.

4 Dactl1 influences

At the time of writing Dactl1 is still at the design stage. However, this stage is nearing completion and we can at least list the major influences which are leading to change. These influences can be classified as follows:

(a) Intensive collaboration with the Dutch Parallel Reduction Machine Project. This has considerably increased our understanding of graph reduction. For example, the soundness and completeness of graph reduction has been demonstrated for weakly regular term rewrite systems[2]. In addition, a new notation for expressing graph reduction has been designed called LEAN (Language East Anglia – Nijmegen). Dactl1 will be directly based on LEAN.
(b) Dactl1 will contain some direct support for the logic variable.
(c) Experience has been gained with the translations of both functional and logic languages. George Papadopoulos has examined the translation of Parlog[3] to Dactl0, and Kevin Hammond has designed a translator from a lazy version of SML[12] to Dactl0. These efforts have provided valuable feedback which has influenced Dactl1 design decisions.
(d) Dactl0 firings for expressing control are very primitive, and consequently hard to reason about. More abstract means of expressing strategy are desirable, but we see research in this area as leading to tools for mapping higher level specifications of strategy onto Dactl0-like markings. Consequently firings will be retained in Dactl1.
(e) Certain features of Dactl0 are inelegant. An example is the need to introduce a user-defined notion of indirection to express simple selector functions, as in section 3.4.1. Although the concept of an indirection node is frequently used in graph rewriting, implementors usually use every possible trick to hide their existence, for reasons of efficiency. Dactl0 at present forces them to be programmed in over-specific detail, and restricts possible implementation techniques. Dactl1 is likely to support more powerful graph rewriting operations which will avoid this.
(f) DACTL0 used a disambiguating meta-rule called *specificity* (described in section 2.4, annexe). Explicit disambiguation using pattern operations will be required in DACTL1.
(g) Module structure in Dactl0 is primitive and needs further elaboration.
(h) Design rules should be developed for translating functional languages into Dactl.

5 Acknowledgements

The considerable contributions of Nic Holt, Mike Reeve, and Ian Watson, who together with the present authors constitute the original Dactl design team are gratefully acknowledged, as is also the support of the Alvey directorate. The comments of the ICL team at Kidsgrove, led by Martin Pixton, proved most helpful. Kevin Hammond made many useful comments on the later drafts of this paper, and helped debug the examples.

ANNEXE: Syntax, operational semantics, and examples of Dactl0

This annexe contains the original specification of Dactl0 as delivered to Alvey in 1985[7]. Some corrections – particularly to the examples which were written without the benefit of a reference interpreter – have been made. However, the editorial intent has been to reveal the state of our understanding at the time the specification was delivered.

1 Dactl0 syntax

This section introduces the formal syntax for Dactl0 and explains the constraints satisfied by a valid Dactl0 program unit.

1.1 Description of Dactl0 syntax

We use a version of BNF to describe the syntax. Terminal symbols are shown in bold face. Alternative productions appear on separate lines. Where it is necessary to continue a production over the end of a line, subsequent lines are indented. Optional parts of productions are enclosed in braces. Parts of productions which may be repeated zero or more times are enclosed in braces followed by an asterisk. A plus character following a matched pair of braces specifies repetition one or more times.

1.2 Character set

A Dactl0 program is represented using the printable ASCII character set with space, tab, and newline characters as separators. The escape sequences used in the C Programming Language[15] may be used to introduce other characters into tokens.

Comments may be included in a Dactl0 program. All characters from a % character to the end of the line are ignored.

1.3 The context-free syntax

The symbol TOKEN stands for any sequence of alphanumeric characters and other printable characters except those used for special purposes in the rest of the syntax. The C Language escape conventions are allowed within TOKENs and may be exploited to allow a TOKEN to be any ASCII character sequence. The case of characters in a TOKEN is significant.

Keywords are shown in bold face, and character case within keywords is not significant. It is possible to construct a TOKEN containing the same characters as a keyword by using the character escapes.

The productions of the syntax are introduced individually and a brief explanation of the purpose of each feature is given.

```
UNIT      ::= PROGRAM
              MODULE
PROGRAM   ::= program TOKEN ; { ITEM }* rewrite { MARKING } TERMREF
              { where DEFS } ; endprogram TOKEN ;
```

A PROGRAM consists of a number of ITEMs which define the set of rules in the graph rewrite system to be modelled. The final element of a PROGRAM describes a marked term which is to be constructed and rewritten according to these rules. Each program is given a name which must be repeated following the endprogram keyword.

```
MODULE    ::= module TOKEN ; { MODITEM }* endmodule TOKEN ;
```

The other form of Dactl0 program unit is a MODULE which defines a set of rules along with the patterns and atomic tokens used in the rules. Each module is given a name which must be repeated following the endmodule keyword. Modules may be imported by other program units but the only features of the implementation visible are those declared public by the defining module.

```
ITEM      ::= imports { TOKEN ; }+
              atom { TOKEN ; }+
              pattern { PATTDEF ; }+
              rule { RULE ; }+
```

The definitions provided in a program or module take one of four forms: the rewrite rules and public tokens defined by a Dactl0 module may be imported; a number of atomic tokens may be declared; tokens representing patterns may be declared; and finally, rewrite rules may be given.

```
MODITEM   ::= ITEM
              public { TOKEN ; }+
```

The constituents of a MODULE are basically ITEMs as defined above. Only the atoms and patterns declared public are made available to a program unit importing the module.

```
PATTDEF   ::= TOKEN = TERM
```

Pattern definitions name patterns which may be used during matching on the left-hand side of a rule. Some patterns can be used to construct terms.

```
RULE        ::= TERM RHS { where DEFS }
RHS         ::= { := TERM } ⇒ { MARKING } TERMREF
                := { MARKING } TERM
                :⇒ { MARKING } TERM
```

The RULEs in a PROGRAM or MODULE are rewrite rules for transforming graphs. The TERM forming the left-hand side of a RULE is a pattern which may match a subgraph of the graph being executed. Execution of rules is described in detail in section 3.

The second and third styles of the RHS of a RULE are syntactic sugaring for versions of the first. A rule of the form lhs := mark trm is equivalent to: lhs := trm ⇒ mark @ (where @ is a special identifier referring to the root of the lhs) while lhs :⇒ mark trm is equivalent to one of the forms lhs := mark trm and lhs ⇒ mark trm, the choice being decided by the implementation.

```
DEFS        ::= DEF { and DEF }*
DEF         ::= IDENTIFIER : TERM
```

Definitions appearing on the left-hand side of a RULE are used to bind IDENTIFIERs to the names of nodes encountered during matching of an enabled term. In an RHS, and in where clauses, IDENTIFIERs are bound to nodes under construction.

```
IDENTIFIER  ::= $ { TOKEN { `TOKEN }* }
                @
                @@
```

For an IDENTIFIER of the first form, no spaces are allowed between the $ and the first TOKEN. The ` is a field-selection operator, whose use is explained below. The special IDENTIFIER @ refers to the root node of the term being re-written. @@ identifies the parent node which causes the current firing of the term being considered.

```
TERM        ::= < { { MARKING } TERMREF }* >
                '{' TERM { , TERM }* '}'
                TOKEN
                IDENTIFIER^
```

The form using braces may be used only on the left-hand side of a RULE, either directly, or implicitly, through use of a defined pattern. It indicates a pattern which will match terms conforming to one of a number of candidate patterns.

The TOKEN form indicates a defined atom or pattern. In the final form, the term denoted is a copy of the root of the term named by the IDENTIFIER.

```
TERMREF     ::= IDENTIFIER
                DEF
                TERM
```

To specify the subterms of a term and the redirection caused by rule application, a reference to a term is required, rather than a term. TERMs may be considered as the nodes in a graph, and TERMREFs as arcs, or references to nodes. When a TERM appears as a TERMREF, a node is to be constructed and a reference to it used.

```
MARKING    ::= !
               ?
```

Markings are attached to references to terms. The marking ! indicates that a firing should be associated with the appropriate arc in the graph referencing the term. The node concerned will therefore be a candidate for rewriting. Absence of a marking indicates that firing will not take place, while ? means it is implementation dependent whether or not the arc carries a firing.

1.4 Patterns and matching

Before a rule can be applied to a fired node, the pattern forming the LHS of the rule must match the subgraph rooted at that node. As matching proceeds, IDENTIFIERs within the TERM representing the pattern are bound to references to the appropriate nodes.

If such an IDENTIFIER appears on the RHS followed by ^, a copy of the node referenced will be required when building terms for overwriting or redirection. This copy reflects the state of the node as encountered during matching. In particular, @^ refers to a copy of the root node when matching starts, and, similarly, @@^ gives a copy of the distinguished parent node.

The IDENTIFIER $ (with no following TOKEN) will match any term found. It may not be used on the RHS but may occur several times on the left-hand side, different occurrences matching different nodes in general.

It is intended that modules should export patterns to provide an implementation-independent means of identifying terms of a given type. An example might be:

```
pattern pair = <cons $hd $tl>;
```

A rule using this might be:

```
rule <head $p:pair> => $p`hd;
```

The module defining pair need not export anything more than the pattern name. In particular, if the atom cons is kept private, there is no mechanism for creating objects of type pair, without using the module. By default, the subterms of a pair term may not be named, so the rule for head above could not be written outside the module. A declaration of the form:

```
public pair; pair`hd; pair`tl;
```

makes the subterms available using the syntax illustrated in the rule for head. However, the actual layout of a pair term is not revealed.

The notation generalises so that, given the following pattern definition:

```
pattern pat = <foo $bar:pair>;
```

if a declaration binds $x to a pat then the subfield $x`bar`hd will denote the $hd field of the $bar field of the node bound to $x. A public declaration may include items such as pat`bar`hd.

If the atom nil has been declared then a pattern introduced by:

```
pattern list = {pair,nil};
```

will match either a pair or nil. The complete term matched by list may be identified, but no field selectors may be used.

Atoms may appear on the right-hand side of rules as may constant patterns. A pattern is constant if it is made up by tupling of subterms which are all either atoms or constant patterns.

1.5 Definitions

Definitions appearing on the left-hand side of a RULE cause the IDENTIFIER to be bound to the root of the corresponding subterm, if matching of the rule is successful. Any other occurrence of an IDENTIFIER on the left-hand side is also, in effect, a defining occurrence. It will match, and be bound to, any node.

Definitions appearing in the RHS of a rule are used to bind the IDENTIFIER to the root of a term under construction. Every IDENTIFIER appearing in a rule must have exactly one defining occurrence. Multiple occurrences of an IDENTIFIER in the RHS of a RULE denote references to a shared term so that true graph structures may be built.

There is no prohibition of recursive definitions. Such definitions will build cyclic graphs. A node must be constructed for each definition and each unnamed tuple. The components of a node formed by tupling are references to the node comprising the subterms. To determine the components of a node formed by copying a referenced node, it is necessary that the components of that node are determined first. In some cases this is not possible, and the RHS is badly formed. A trivial example of an illegal DEF would be:

```
$a:$a^
```

1.6 Modules

The module structure in Dactl0 is an attempt to address the need for partitioning of software development. It is also intended that the module

mechanism should be used to encapsulate special hardware mechanisms and implementation techniques which are not of concern at the Dactl level.

By providing standard interfaces in the form of Dactl0 rules, atoms, and patterns, the actual implementation of the lowest level of facilities may be hidden. Conceptually, a standard module declares patterns such as int, real and boolean in the normal way. Constant values are just atoms and the patterns are simply the union of the appropriate atoms. The boolean pattern might be declared by:

```
atom true; false; pattern boolean = { true, false };
```

All the rules defined in a module and any it imports are exported, but a program cannot create terms which will match these rules unless it can use the atom and pattern symbols involved. Atom and pattern tokens to be made available outside a module must occur in a public declaration. Tokens imported from other modules remain hidden unless re-exported. All the tokens used in a program unit must be declared as atoms or patterns in the unit, or declared public in an imported module.

The design of the Dactl0 module structure is orthogonal to the main design issues in the language. It does not reflect the best research in this area and it is hoped that suggestions for improvements will be forthcoming.

1.7 The complete syntax

```
UNIT        ::= PROGRAM
                MODULE
PROGRAM     ::= program TOKEN ; { ITEM }*
                rewrite { MARKING } TERMREF { where DEFS } ;
                endprogram TOKEN ;
MODULE      ::= module TOKEN ; { MODITEM }* endmodule TOKEN ;
ITEM        ::= imports { TOKEN ; }+
                atom { TOKEN ; }+
                pattern { PATTDEF ; }+
                rule { RULE ; }+
MODITEM     ::= ITEM
                public { TOKEN ; }+
PATTDEF     ::= TOKEN = TERM
RULE        ::= TERM RHS { where DEFS }
RHS         ::= { := TERM } ⇒ { MARKING }
                TERMREF := { MARKING } TERM
                :⇒ { MARKING } TERM
DEFS        ::= DEF { and DEF }*
DEF         ::= IDENTIFIER : TERM
IDENTIFIER  ::= $ { TOKEN { `TOKEN }* }
                @
                @@
TERM        ::= < { { MARKING } TERMREF }* >
                '{' TERM { , TERM }* '}'
                TOKEN
                IDENTIFIER^
```

```
TERMREF     ::= IDENTIFIER
                DEF
                TERM
MARKING     ::= !
                ?
```

2 Operational semantics of Dactl0

In this section we describe an execution model for Dactl0.

2.1 Expression graphs

A Dactl0 program defines a graph rewriting system (GRS) together with an expression graph to which the rewrite rules are to be applied.

Ignoring markings for the moment, a TERM represents a directed graph. ATOMs may be thought of as leaf nodes, and tuple TERMs as interior nodes. An arc proceeds from each interior node to each of the nodes represented by the components of the tuple. Sharing – i.e. arcs pointing to the same node – is expressed by repeated use of the same IDENTIFIER.

2.2 The three-clouds model

As a graph is executed, firings may be attached to or removed from the arcs. The presence of a firing on an arc from a node N to a node N′ signifies that N requires N′ (more precisely, the subgraph rooted at N′) to be further executed. N′ will eventually be executed and will signal to N that some work has been done by removing the firing from the arc. We will also speak of N′ being "fired" by N.

The RULEs in a Dactl0 program specify how firings are attached and removed by means of the markings on their right-hand sides. The markings are considered to be attached to arcs of the graph. The presence of a ! against a TERM T which is an immediate subterm of a TERM T′ signifies that when this arc is created by an execution of the rule, it is to be fired.

The Dactl0 programmer may use the marking ? when he does not wish to specify whether the arc is to be fired. When a rule is executed it must not contain ?'s – the choice must have been resolved by then.

To explain graph execution we partition the nodes of the graph into three classes, or "clouds". As computation proceeds, nodes move between the clouds. The three clouds are called Inactive, Enabled, and Blocked.

A node is Inactive when it has not been fired by any of its parents. No-one is currently demanding any work to be done on that node, and it is not considered for rule-matching.

When a node is fired, the machine tries to match the subgraph rooted at that

node against the left-hand side of each rule. If any pattern matches, the node becomes Enabled. The rule whose left-hand side matched (or any one of such rules, if more than one matches) will be executed. For a rule of the form

$$P := O \Rightarrow R$$

if the pattern P matches the subgraph rooted at node N, then the graphs O and R are constructed, and the root of O overwrites N. Arcs in O or R marked with ! are fired. That is, the nodes they point to become candidates for rule matching and execution. The arc along which N was fired is redirected to point to the root of R. That arc receives any firing attached in the rule to the root of R. If it is not fired then the firing parent of N is notified that its demand has been satisfied. That parent is then once more a candidate for rule matching.

Note that the only parts of the graph which are updated when a rule is executed are the root of the subgraph matched by the left-hand side of the rule and the arc along which it was fired. The only other effect of rule-execution is to create new nodes and arcs.

When a node is fired it is possible that no rule matches. The node becomes Blocked. It will only be reconsidered as a root node for rule matching if it has at least one fired arc to an immediate descendant, and (as a result of applying rewrites to that descendant) the firing is later removed.

When a node is Blocked, it is possible that some indirect descendant of that node may get overwritten in such a way that an attempted rule match at the Blocked node would now succeed. Nevertheless, the node remains Blocked, even if some other node attempts to fire it. Thus a Blocked node with no fired arcs to any of its descendants must remain Blocked forever, or, as we may term it, Dead.

We might invoke some default action to handle this case, causing a Dead node, for example, to fire all of its descendants. However, we exclude "default options" from Dactl0, since it is not clear that there is a uniformly sensible default action. For example, the above suggestion might merely replace deadlock by livelock. The provision of defaults is the responsibility of the Dactl0 programmer or his programming tools.

This description is summarised in Fig. 2. C stands for node creation, F for a node being fired, R for redirection of an arc to a descendant, MX for matching and execution of a rule, and $\sim$M for failure to find any matching rule.

2.3 Matching

The patterns which may occur on the left-hand sides of RULEs are a subset of the TERMs, those which are constructed only from ATOMs, IDENTIFI-

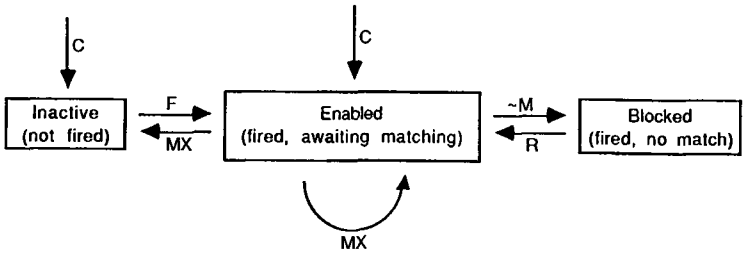


Fig. 2

ERs, and tupling, and which satisfy certain other constraints. An ATOM matches only itself. A tuple matches a node whose descendants match the respective components of the tuple.

An IDENTIFIER of the form $ or $x is a "wild card", matching any node. In the case of $x, the same IDENTIFIER can be used on the right-hand side to refer to the same node. $ may be used when the programmer does not wish to refer to the node again. An IDENTIFIER $x cannot be repeated on the left-hand side, and if different identifiers $x and $y appear, it is possible for them to be bound to the same node. Thus the matching process is not influenced by the presence or absence of sharing in the graph. There is no identity test on nodes. The reasons for this are more fully explained in [6].

The IDENTIFIERs @ and @@ may only appear in TERMs in the RHS of a RULE. @ is a reference to the node matched by the root of the pattern on the left-hand side. @@ is a reference to that parent of @ which fired it, provoking execution of the rule. Note that the TERM @^, used in an RHS, denotes the contents of the root of the matched subgraph at the time of matching (i.e. before performing the overwrite).

The overwrite portion of a RULE must specify the contents of a node, while the redirection part must specify a reference to a node. This accounts for the slight differences in their syntax. An identifier is a reference to a node, and must therefore be dereferenced by the ^ operator if it is to appear as an overwrite. (Another method is illustrated in the functions head and tail in the lists module example below.) A tuple represents the contents of a node; its use as a TERMREF implies the creation of a node to hold it.

A pattern is only allowed to match a subgraph containing fired arcs in a very restricted way. An arc has been fired because its source node has requested its target node to be further evaluated. It is inappropriate to look at the contents of the target while this is happening. For this reason, the only cases where markings can usefully occur are on the anonymous identifier $, since whatever the state of the arc concerned, its existence, which is all that $ detects, cannot be in doubt.

Thus $, like $x, does not match a node at the end of a fired arc. !$ matches only such nodes, and ?$ matches either.

A rule which performs an overwrite, and not just a redirection, may not have a left-hand side able to match a subgraph whose root has outgoing fired arcs. An overwrite in effect cuts all the outgoing arcs of the root. Only the source of an unfired arc needs to know of its existence, but a fired arc is also known to its target. Cutting a fired arc would require notifying its target node, and in a system as loosely coupled as we envisage running Dactl, this could pose difficulties.

The restrictions described above are made in the interests of flexibility of possible implementations and the desire to retain good mathematical properties of the execution model, rather than because we believe that implementation of Dactl0 might otherwise be difficult. Indeed, most of the implementation methods that the authors can imagine would, for most of the restrictions, have no such difficulty. But we do not wish to lift them until we discover strong reasons for requiring greater expressive power. The reader is encouraged to provide us with such reasons.

2.4 Ordering of rules

Rather than assuming that rules are considered in the order in which the programmer gives them, it is convenient to require that rules with more specific patterns will be tried before those with less specific patterns. That is, if the set of graphs which one rule is capable of matching is a superset of those which another rule can match, then the latter rule will be tried before the former. This defines a partial ordering on rules; an implementation may or may not refine this to a total ordering. Later versions of Dactl may allow the programmer to override or modify the specificity ordering.

2.5 Parallel execution

Parallel execution of rewrite rules raises questions of exclusive access by processing agents to parts of the graph. A Dactl0 pattern can be arbitrarily large. If we were to assume that every processing agent acquires exclusive access to the whole graph when attempting a pattern match, much potential for parallelism would be lost. If we instead allowed an agent to perform matching by acquiring exclusive access to the relevant subgraph one node at a time, this could easily result in deadlock. It turns out, however, that pattern matching can be performed on the basis of much weaker mutual exclusion postulates. We assume that:

(a) An agent wishing to attempt rule-execution at some node may acquire exclusive permission to do so. This does not exclude any other agent from reading the contents of the node, only from attempting pattern matching rooted at that node.
(b) Reading a node, overwriting a node, reading an arc, and overwriting an

arc are atomic operations. We do not assume that arcs are represented as pointers to their target nodes, contained in their source nodes. They might (or might not) be separate objects: nodes would contain references to arcs, and arcs references to nodes.

An agent with these capabilities can inspect the subgraph rooted at a node, one node or arc at a time. When it discovers that some pattern matches, it then constructs the overwrite and redirection graphs. The original node is overwritten and released by the agent, and the redirection graph is linked to the node which fired the node being matched.

Since the agent never has simultaneous access to all the nodes of the subgraph being matched, it is possible that while it is exploring that graph, other agents are changing it. The nodes which it sees may in fact never have existed simultaneously. How then, can we reason about this method of execution of GRSs? The problem is less serious than it may appear. We can show that for a system of Dactl0 rules generated by translation into Dactl0 of a conventional rewrite system satisfying certain conditions (such as those of Hoffmann and O'Donnell[8] or Staples[16]), executing a graph by Dactl0 will give the same result as the original rewrite system. We are investigating extensions of this result to more general systems of rewrite rules.

3 Examples

For comparison, some of the following examples of Dactl0 are also written in SASL[17].

3.1 A simple example

SASL:

```
fib 0 = 1
fib 1 = 1
fib n = fib(n − 1) + fib(n − 2)
```

Dactl0:

```
program fib;
imports arithmetic;
atom fib;
rule
<fib 0> := 1;
<fib 1> := 1;
<fib $n:int> := !<+ !<fib !<- $n 1>> !<fib !<- $n 2>>>;
<fib $n> := !<fib !$n>;
rewrite !<fib <fib 5>>;
endprogram fib;
```

Note that Dactl0 requires the fourth rule to deal with the case where the argument to fib is not yet in the form of an integer and needs further evaluation.

3.2 *Illustration of control over evaluation order*

SASL:

```
head a:b = a
tail a:b = b
listeval () = ()
listeval a:b = scons a (listeval b)
scons a () = a:()
scons a b:c = a:b:c
```

Listeval is the identity function on lists, but forces the whole of the list to be evaluated. In Dactl0 we can define two versions of this function. Seqevallist expands the whole list, and replies when this has been done. Parevallist also expands the whole list, but replies as soon as the list has been brought to the form of nil or a cons. The remainder of the list continues to be evaluated in parallel.

Dactl0:

```
module lists;
atom nil; cons; head; tail; scons; seqevallist; parevallist;
pattern pair = <cons $hd $tl>; list = {nil,pair};
public nil; cons; pair; list; head; tail; seqevallist; parevallist;
rule
nil ⇒ @;
<cons ?$ ?$> ⇒ @;

<head <cons $h ?$>> ⇒ !$;
<head $x> := !<head !$x>;

<tail <cons ?$ $t>> ⇒ !$t;
<tail $x> := !<tail !$x>;

<seqevallist nil> := nil;
<seqevallist <cons $h $t>> := !<scons $h !<seqevallist $t>>;
<seqevallist $x> := !<seqevallist !$x>;

<scons $h $t> := <cons $h $t>;

<parevallist nil> := nil;
<parevallist <cons $h $t>> := <cons $h !<parevallist !$t>>;
<parevallist $x> := !<parevallist !$x>;

endmodule lists;
```

We can also define a version of the function which not only expands the list structure, but also fires the elements of the list. All that is necessary is to add another firing to the second rule for parevallist:

```
<parevallist <cons $h $t>> := <cons !$h !<parevallist !$t>>;
```

3.3 *Quicksort*

SASL:

```
quicksort () = ()
quicksort (h:t) = split h t () ()
```

```
split x () a b = append (quicksort a) (x:(quicksort b))
split x (h:t) a b = (x < h) -> (split x t a (h:b))
                           ; (split x t (h:a) b)
append () 1 = 1
append (h:t) 1 = h:(append t 1)
```

Dactl0:

```
program quicksort;
imports lists;
imports logic;
imports arithmetic;
atom quicksort; split; append;

rule
<quicksort nil> := nil;
<quicksort <cons $h $t>> := !<split $h !$t nil nil>;
<quicksort $x> := !<quicksort !$x>;

<split $x nil $a $b> := !<append
                            !<quicksort !$a>
                            <cons $x !<quicksort $b>>
                        >;
<split $x <cons $h $t> $a $b> := !<ifthenelse !<lt !$x !$h>
                                      <split $x $t $a <cons $h $b>>
                                      <split $x $t <cons $h $a> $b>
                                  >;
<split $x $y $a $b> := <split $x !$y $a $b>;
<append nil $x> => !$x;
<append <cons $h $t> $x> := <cons $h !<append $t $x>>;
<append $a $b> := !<append !$a $b>;

rewrite !<quicksort <cons 3 <cons 4 <cons 1 <cons 5 <cons 2 nil>>>>>>;
endprogram quicksort;
```

The Dactl version is deliberately only partly lazy. The recursive calls of quicksort in the definition of split are both fired. The result is that with unbounded parallelism, the list is on average sorted in time proportional to the logarithm of its length.

3.4 Higher-order functions

SASL:

```
twice f x = f (f x)
```

Dactl0:

```
<<twice $f> $x> := !<!$f <$f $x>>;
<twice $x> => @;
twice => @;
<$x $y> := !<!$x $y>;
```

Here is an example. We assume that succ is the successor function on integers. Blocked nodes are indicated by a #.

```
!<<twice <twice succ>> 0>
:= #<!$f <$f 0>> where $f: <twice succ>
:= !<$f <$f 0>> where $f: <twice succ>
:= #< !succ <succ <<twice succ> 0>>>
:= !<succ <succ <<twice succ> 0>>>
:= #<succ !<succ <<twice succ> 0>>>
:= #<succ #<succ !<<twice succ> 0>>>
:= #<succ #<succ #<!succ <succ 0>>>>
:= #<succ #<succ !<succ <succ 0>>>>
:= #<succ #<succ #<succ !<succ 0>>>>
:= #<succ #<succ !<succ 1>>>
:= #<succ !<succ 2>>
:= !<succ 3>
⇒ 4
```

3.5 Bottom-avoiding merge

Bottom-avoiding merge is a non-deterministic function of two lazy lists. It produces a list which is some merging of the two lists. If one of the two lists produces its first element and attempted evaluation of the other never yields any of its elements at all, then the bottom-avoiding merge is required to yield the elements of the list which could be evaluated without waiting for evaluation of the other. The classic example where this is required is the merging of input streams from two terminals. From within the programming language a terminal can be regarded as a lazy list, which turns into an actual list of characters as and when someone types on the terminal's keyboard. Attempting to evaluate the head of such a list makes the program wait until something is typed. It is a non-trivial problem in parallel declarative programming to ensure that a program attempting to read from each of two terminals is able to use input from either, even if nothing is ever typed on the other, and to do this without busy waiting.

Bottom-avoiding merge cannot be expressed in SASL.

```
module bam;
imports lists;
atom bam; bam0; bam1;
public bam;
rule
<bam $x $y> := !<bam0 !<bam1 !$x !$y $x $y>>;
<bam0 <bam1 $ ?$ $x:pair $y>> :=
<cons $x`hd <bam0 <bam1 !$x`tl !$y $x`tl $y>>>;
<bam0 <bam1 ?$ $ $x $y:pair>> :=
<cons $y`hd <bam0 <bam1 !$x !$y`tl $x $y`tl>>>;
<bam1 $ ?$ pair $> ⇒ @;
<bam1 ?$ $ $ pair> ⇒ @;
<bam1 $ ?$ $x $y> ⇒ !<bam1 !$x !$y $x $y>;
<bam1 ?$ $ $x $y> ⇒ !<bam1 !$x !$y $x $y>;
endmodule bam;
```

The last two rules for bam1 could be omitted if the programmer guarantees that bam will only be applied to list expressions which, when fired, do not reply until they have turned into the form of a pair. For simplicity, the two

arguments are assumed to be infinite lists, but the definitions could easily be extended to deal with finite lists as well.

3.6 Assignment

Dactl0 can model non-declarative concepts such as assignment. We can program a cell with read and write operations on it. The cell uses @@ to find out what operation is being performed and in the case of a write, to read the value being written.

```
module cell;
atom cell; cell1; read; write; write1; swap; swap1;
public read; write; swap; cell;
rule
<cell $n> := !<cell1 $n @@>;
<cell1 $n <write1 !$ $m>> := <cell $m>;
<cell1 $n <swap1 !$ $m>> := <cell $m> ⇒ !$n;
<cell1 $n $> := <cell $n>;

<read <cell $n>> ⇒ !$n;
<read $c> ⇒ !<read !$c>;

<write $c:<cell $n> $m> := !<write1 !$c $m>;
<write $c $m> := !<write !$c $m>;
<write1 $c:<cell $> $> ⇒ $c;

<swap $c:<cell $> $n> := !<swap1 !$c $n>;
<swap $c $n> := !<swap !$c $n>;
<swap1 $c $n> ⇒ !$c;

endmodule cell;
```

3.7 Semaphores

Semaphores are a tool with which one may implement mutual exclusion constraints on concurrent processes[5]. We can program a semaphore in Dactl0 and use it to ensure that certain terms are never fired simultaneously.

```
module sema;
atom mutex; go; done; sema; sema1; sema2;
public mutex; sema; done;
rule
<mutex $x $y> := !<go !$x $y>;
<go <sema> $y> ⇒ !$y;
<sema> := !<sema1 @@>;
<sema1 $x:<go ?$ $y>> := !<sema2 $x !$y>;
<sema2 $ $> := <sema>;
endmodule sema;
```

We illustrate the use of this module by the following example, which uses the cell module given earlier:

```
<!<mutex $s $x1> !<mutex $s $x2>>
      where $s:<sema>
      and   $x1:<read <write $c 1>>
```

```
and    $x2: <read <write $c 2>>
and    $c: <cell 0>;
```

$x1 and $x2 are, in effect, two processes, each of which writes an integer to the cell $c and then reads it. The semaphore ensures that $x1 and $x2 are never simultaneously in the fired state. Thus each of the processes will read the same value it wrote. It is impossible for the two processes to both write, and then both read, as can happen with the term:

```
<!$x1 !$x2>
      where $x1: <read <write $c 1>>
      and   $x2: <read <write $c 2>>
      and   $c: <cell 0>;
```

References

1 BAETEN, J.C.M., BERGSTRA, J.A. and KLOP, J.W.: 'Priority rewrite systems'. Report CS-R8407, Department of Computer Science, Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam.

2 BARENDREGT, H.P., VAN EEKELEN, M.C.J.D., GLAUERT, J.R.W., KENNAWAY, J.R., PLASMEIJER, M.J. and SLEEP, M.R.: 'Term Graph Rewriting'. University of East Anglia, October, 1986.

3 CLARK, K. and GREGORY, S.: 'PARLOG: Parallel Programming in Logic'. ACM TOPLAS, Vol. 8, No. 1, Jan., 1986.

4 DARLINGTON, J. and REEVE, M.: 'ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages'. Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire, Oct., 1981.

5 DIJKSTRA, E.W.: 'Cooperating sequential processes'. In: Genuys, F. (editor), 'Programming Languages', Academic Press.

6 GLAUERT, J.R.W., HOLT, N.P., KENNAWAY, J.R., REEVE, M.J. and SLEEP, M.R.: 'An active term rewrite model for parallel computation'. Internal Report, School of Information Systems, University of East Anglia, Feb., 1985.

7 GLAUERT, J.R.W., HOLT, N.P., KENNAWAY, J.R., REEVE, M.J., SLEEP, M.R. and WATSON I.: 'DACTL0: A Computational Model and an associated Compiler Target Language' Internal Report, School of Information Systems, University of East Anglia, May, 1985.

8 HOFFMANN, C.M. and O'DONNELL, M.J.: 'Implementation of an interpreter for abstract equations'. In: ACM Conference on Computer Science, 1983.

9 HEWITT, C.: 'Viewing control structures as patterns of message passing'. Artificial Intelligence, Vol. 8, 1977, 323–364.

10 HUET, G. and OPPEN, D.: 'Equations and rewrite rules: a survey'. Report CSL-111, SRI International, 1980.

11 KLOP, J.W.: 'Combinatory Reduction Systems'. Mathematical Centre Tracts 127, Kruislaan 413, 1098 SJ Amsterdam.

12 MILNER, A.J.R.G.: 'A Proposal for Standard ML'. Internal Report CSR-157-83, CSD, University of Edinburgh.

13 O'DONNELL, M.J.: 'Equational Logic as a Programming Language'. MIT Press, 1985.

14 REDDY, U.: 'On the relationship between logic and functional languages'. In: DeGroot, D. and Lindstrom, G. (editors), 'Functional and Logic Programming', Prentice-Hall, 1985.

15 RITCHIE, D.: 'C Reference Manual'. Bell Telephone Laboratories, Murray Hill, 1974.

16 STAPLES, J.: 'Optimal evaluations of graph-like expressions'. Th. Comp. Sci., Vol. 10, 1980, 297–316.

17 TURNER, D.A.: 'SASL language manual'. University of St. Andrews, 1976.

18 WATSON, I. and GURD, J.: 'A Practical Data Flow Computer'. IEEE COMPUTER, Vol. 15, No. 2, Feb., 1982.

PARALLEL DECLARATIVE SYSTEMS

Hardware parallelism makes it possible to achieve high performance economically by exploiting the availability of cheap, high-powered microprocessors, themselves made possible by advances in VLSI technology. This final section describes two parallel systems, both using the principle of graph reduction, and the system software needed to support one of these.

Designing system software for parallel declarative systems

P. Broughton, C.M. Thomson, S.R. Leunig and S. Prior
(Members of Flagship System Software Team) ICL Mainframe Systems, West Gorton, Manchester

Abstract

The aim of this paper is to give an overview of the work being carried out by Flagship project to design system software for parallel hardware. The software will provide a computing environment within which programs which are mainly declarative can be designed and run. The paper is a snapshot of the early stages of the design process. It discusses some of the issues being addressed and our approach to design.

1 Introduction

Hardware which will exploit the inherent parallelism of declaratively expressed programs and applications opens up the exciting prospect of almost limitless performance[1,2]. The larger the application or the more users a system has, the greater the potential for parallelism and the greater is the number of parallel processors which can devour the work. At the same time declarative languages allow the programmer to express his problem to the computer without the confusion of worrying simultaneously about how to make the computer solve it[3]. However, no matter how powerful the hardware or attractive the languages, these systems will be used only by the devout few unless they are capable of providing an attractive programming environment, of managing themselves and their resources to a level expected in the 1990s and of adaptation to meet the rapidly advancing ideas and languages of the declarative world. The responsibility for all these lies with the system software.

Our research will take advantage of ICL's existing VME operating system and also of UNIX by adopting a policy of hosting FLAGSHIP with these systems. By this means we shall avoid designing a new operating system and be able to concentrate on the issues posed by the declarative parallel system; so producing code which is sufficiently robust and timely to be used by other researchers.

This strategy of hosting will allow us to produce more quickly, systems for research which are largely or even completely running on the sequential host.

Tactically however, we must first understand the design of the system targetted to our parallel hardware and develop a minimal system to run on our early prototypes. This is a vital contribution to the design and validation of the FLAGSHIP hardware.

The structure of Flagship overall system is given in Fig. 1. The architectural style of the system as viewed by language users and compilers is guided by the Programming Reference Model (PRM). A concrete realisation of the ideas contained in the PRM will be the Common Machine Interface (CMI); the provision of this interface on target parallel hardware and under both VME and UNIX is the task of the system software. This consists of a "library" of system functions in the parallel hardware plus functions provided by the host. The host also provides system management. The PRM is discussed in more detail later in this paper.

Besides provision of the CMI the system software has other duties such as management of system resources and provision of a secure multi-user environment. A great majority of these duties require support from the hardware or underlying system and will be defined in the Implementation Specific Interface (ISI) particular to the parallel hardware or UNIX/VME host system. These issues are covered in more detail later.

For the system software ultimately to run efficiently on parallel hardware it must itself exploit the parallelism available. We must therefore implement our code in a declarative style. Declarative languages are designed so that the programmer does not need to think about the state of his program at any point during its execution. This lack of control of state in the languages and

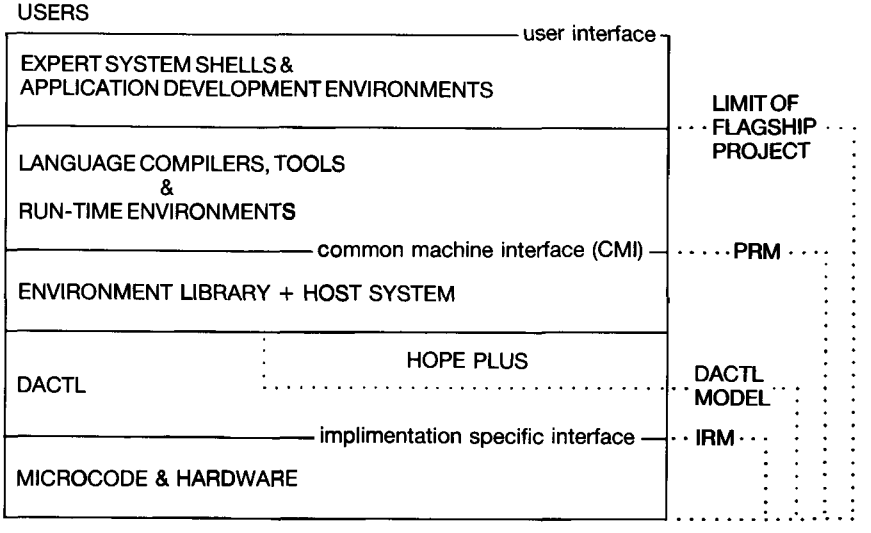


Fig. 1 Overall Flagship system

their use in writing system software (where the task is to control hardware and process state) presents another problem to be solved by the team. This is addressed by our choice and design of implementation languages and by the Declarative Alvey Compiler Target Language (DACTL).

To begin a more detailed discussion of all these issues let us consider our approach to getting a correct design of this first ever declarative, parallel operating system.

2 The design approach

The principal aim of the design methodology is to adopt a rigorous approach to the development of the system software. Such a strategy places emphasis on the design and specification stages of the software life cycle. The intention is to reduce the number of design faults – the most expensive to repair – and to provide a solid foundation of the rest of the development cycle.

The approach being adopted has the following characteristics:

- Formal specification
- Executable model in declarative style
- Transformation of specification and program
- Fagan inspection

The method of creating formal specifications of the major Flagship components is relatively untried outside academic circles and needs a careful approach to ensure expectation does not outweigh investment. The feasibility of using the Vienna Development Method (VDM)[4] is being investigated.

VDM embodies the concept of transformation (called reification in VDM) to progress from abstract specifications to concrete implementations. Each transformation is accompanied by an operation called a retrieve function. The retrieve function is used to demonstrate that all the information in the abstract version is still available in the more concrete form, i.e. correctness has been preserved.

The main components of the system software are being modelled using the functional language Hope, to provide rapid feedback on ideas and establish a declarative style for implementing state-based software systems. Initially the modelling activity will precede formal specification, so that some practical degree of confidence is achieved before devoting effort to more theoretical issues. It is expected that as experience grows, a formal treatment will be the first step in the development cycle. However, in practice the specification and modelling activities should be viewed as iterative rather than sequential steps.

Program transformation is a feature of declarative language research at

Imperial College as part of the Flagship project. It is hoped to exploit this research by making it part of the systems software development route. As with VDM, the idea is to use meaning-preserving transformations to produce efficient programs from inefficient ones. These techniques are in their infancy, but eventually it should be possible to use them to help transform our Hope models into efficient programs.

The specifications and models will be subject to Fagan-style inspections to provide further verification of the design activities. Inspecting formal specifications is a more precise process than inspecting natural language documentation, hence inspections should be more effective.

The design of the Flagship system software is presented as a layered architecture, commonly represented as an onion-ring structure or a tower structure. Each layer is constructed using facilities offered by lower layers. The innermost layer uses facilities provided by the hardware. The interfaces between layers are collections of *abstract data types.*

An abstract data type (ADT) consists of a data type description together with a set of operations; these allow the user to create objects of the specified type which can be manipulated using the operations provided. For example, a STACK could be represented as an ADT with operations like PUSH and POP to manipulate it. Thus an ADT hides irrelevant implementation details from the user. For example, the user of a STACK does not need to know whether it is implemented as a list or a vector.

A pilot implementation of the system software will be developed to run on a software simulation of the Implementation Reference Model (IRM) – the hardware architecture – running on a host system. This simulator will enable some of the design to be validated before Flagship hardware is available. The pilot system will play an important role in validating the IRM and in establishing our design methods but will necessarily fall a long way short of the target PRM implementation.

3 The Programming Reference Model (PRM)

The Programming Reference Model (PRM) provides a common framework for language systems and applications, capturing the architectural philosophy of the Flagship system. It aims to provide a common set of high-level concepts (and associated mechanisms) which can be used by all language and application environments, so that interworking between applications and between software components constructed using different languages is facilitated; as a consequence, it must specify in abstract form a large part of the user interface to the Flagship system software. The high-level concepts chosen to be represented in the PRM are intended to enable application programmers and designers to take a more abstract view of application design than in third and fourth generation systems, so that the cost of application development is reduced.

3.1 Objects and transactions

The PRM supports a largely declarative style of programming, but not a totally declarative one; the view is taken that in a world where the majority of applications include amongst their objectives the update of large shared databases the architecture should cater for this directly, rather than conceal the issues of update and shared storage behind an obscure functional programming trick like stream processes[5]. The mechanism for looking at and updating an object should be the same whether the object is in main store or on a disc or other mass storage mechanism, so that application developers need not concern themselves with the mechanics of handling mass storage. Non-declarative actions require synchronisation, which would severely impair the effectiveness of parallel working if it occurred frequently, so the PRM must arrange for these actions to be packaged up into large-grain operations. Also, uncontrolled updating of store makes it very difficult to reason effectively about program behaviour. Update of shared data introduces non-determinism into the system, since the result of an operation will depend on when it happens to be done, and it is necessary to constrain this non-determinism to avoid unwanted results; interactions between programs operating on shared data require a concept of atomic action so that consistency of the shared data can be maintained and partially updated versions of the data are avoided. Rather than provide low level mechanisms like explicit locks or semaphores to address these issues, the PRM provides a single high level concept: a (typed) persistent object store[8] which can be interrogated and updated only by an atomic transaction operation. As well as atomicity of transactions, non-determinism can be limited by use of the Strict operator which when applied to a function forces evaluation of an argument before application of the function, so that the order in which operations are carried out is fixed by the order of functional composition.

Atomicity of transactions requires that the evaluation of the various parameters to the transaction proceeds sufficiently far that all non-determinism is resolved before the transaction is committed; and that where a transaction makes multiple references to the same object it will see the same object value each time. As a result evaluation of object store references within the arguments to a transaction operation will not commence until the transaction is started, but computation must then proceed at least far enough to resolve any non-determinism. The PRM takes no view on whether computation within the transaction should continue beyond the minimum required to eliminate non-determinism; a Normalise operator is provided to force reduction to normal form (i.e. to ensure that all computation of new values is completed) before an update is performed.

Nesting of transactions is permitted in order that the provider of an abstract interface can use transactions within his program independently of the context (possibly within a transaction) in which the interface is invoked. A nested transaction commits its result independently of any enclosing transaction, while an enclosing transaction may depend on values returned to it by

nested transactions. No nested transaction may update any object visible to any enclosing transaction, else the consistency of the object store seen by the enclosing transaction would be violated. The updates (if any) derived in an enclosing transaction do not affect the object store seen by a nested transaction. The "nested top level transaction" of [12] is provided, but PRM terminology does not call this case "nested". The coordinated commitment form of nested transaction (called Dependent Sub-Transaction[13]), where subtransaction results are visible within the enclosing transaction but are not committed until the outermost transaction completes, is not provided. Coordinated commitment of transactions which are necessarily separate, and hence not nested (as arises in a federated system[14]), is achieved by communication at the "secure" phase of the transactions.

3.2 Types, values and expressions

To facilitate mixed language working the PRM specifies the base values and value constructors available in the system, together with a very general type system. The type system aims to be sufficiently general to cope with a wide range of language-level type systems, and provides universal polymorphism[9] both as subtype (set inclusion) polymorphism (i.e. a value can have many types) and as parametric polymorphism (e.g. a function can operate on arguments of different types by having extra arguments to specify the types of the "real" arguments). Types are first class values, and both a universal type and the type of all types are provided. Values are self-typing and there are operations to convert from implicitly typed values to explicit dependant pairs. The structural aspects of typing can be checked by the system, so that high level language compilers can invoke the PRM type system to check parameters/results at module boundaries. Base types include integer, "real" (floating point), character, string, truth-value, type (type of all types), value (universal – everything is of this type), and unit (type containing only a special distinguished value). Type constructors include:

- product, function, enumeration, set, array, fixed-point,
- map (finite function),
- union (disjoint, i.e. the values are labelled to say which component of the union they come from),
- dependent product (for pairs the type of whose second component depends on the value of the first),
- dependent function (for functions where the type of the result depends on the values of the arguments),
- dependent map.

The last three type constructors form the basis for parametric polymorphism instead of the more usual quantifers. Record types, lists, and ordinary unions can easily be constructed from this set. The enumeration construction permits any constructible set of values to be treated as a type, that is the values in an enumeration type are not just formal labels as in many systems – label enumerations can be built as disjoint unions of the unit type with itself.

Consequently the type system is more expressive for inheritance relationships than that of Cardelli[11]. Like any type system of similar power, the PRM type system is undecidable.

As well as straightforward values, the PRM allows names to be constructed and expressions containing them to be manipulated, and provides mechanisms for binding names and building contexts; the name constructor can construct names of any type. Expressions also arise in the context of object-values, since anything which refers to operations is necessarily an expression rather than a pure value and the only way one object-value can depend on other object-values is by referring to the operations which look them up.

The value construction system includes a method for constructing error values, with system defined error values embedded in each type; erroneous computations will in general deliver error values rather than result in the computation being aborted, whether the error arises from a fault in application software, system software, or a hardware component. These error values are first class values and may be manipulated in much the same way as any other values, so that resilience to various classes of failure may be achieved by in-line programming rather than by complex asynchronous exception handling as in conventional error handling systems like that of ML or Ada.

3.3 Protection

Static access controls similar to those found in any modern operating system (e.g. to protect one user's files from improper use by other users) are defined, based around the idea of permission for a "principal" (the system representation of a person accountable for his or her actions) to use an object in various ways; this is augmented by dynamic security provision based around the US DoD "orange book" standard[15] (and capable of achieving certification to at least B2 level) since it is estimated that any serious commercial (e.g. banking) application will require this level of security in Flagship timescales. Security is treated as an architectural issue since attempts to build a secure system on top of an architecture which does not address security are likely to fail.

3.4 Processes

Explicit parallelism is a concept needed by many applications. Processes can fire off subprocesses or new processes, and specify appropriate scheduling, budgeting, and security constraints for them. Since the new process (or subprocess) can share values with the original process, inter-process communication using shared lists is available; the processes can also communicate using shared objects and the transaction mechanism; so no other concept of inter-process communication is needed in the PRM. Within a (sub-)process the implicit parallelism of the underlying graph-reduction machine applies. Thus the process and transaction concepts are completely orthogonal, so that a transaction may be carried out by an arbitrary set of process nests, while a process may contain an arbitrary set of transaction nests.

3.5 Further work

Much work remains to be done on the PRM; many issues have not yet been addressed; evaluation by users of the PRM (and its implementation in Flagship system software) may lead to some surprises; work on a formal description of the PRM has not progressed far – we have neither a formal mathematical model for the value/type component (but Cartwright[10] is a reasonable starting point) nor a mathematical description for the computational model which can handle the controlled non-determinism of the object store and transactions. However sufficient is known about the major issues to allow design of the Flagship system software to proceed – and this design process will in turn help us to complete the PRM definition.

4 Design issues

While provision of an introductory system running on current sequential hardware will be an important factor in obtaining research feedback, the design of the target parallel Flagship system must be the basis for commercial products into the future and this paper will concentrate on this system. In order for these products to be competitive they will need to be able to provide facilities equivalent to those of conventional systems. The objectives for the target parallel system are as follows:

- It will be a general purpose, multi-user system, either in the form of a hosted system under VME and UNIX, or as a free-standing system.
- It will provide the PRM/CMI as the major interface between the language systems and applications and the underlying system.
- It will provide security at least to the "DoD B2" standard.
- It will be scalable over a wide range of configuration sizes (say 8 to 100 processing elements initially, but potentially many more).
- It will provide the basis of a wide range of system types, e.g. from a single user system to a large multi-user system providing large databases.
- It will provide the basis for high availability systems.

Clearly these are very ambitious objectives and the initial systems will be only steps on the way. In particular a host will be required to provide most of the system management functionality. The need to move from the initial system to the target system imposes one of the most demanding requirements on the system software; it must allow for both the movement of functionality from the host to the Flagship machine, and for the enhancement of the system software to provide extra facilities, and especially to allow the tracking of the changes to the hardware.

So far very little research has been done on the design and implementation of operating systems for highly parallel graph reduction systems such as Flagship. This means that there are a number of important design issues which will need to be addressed during the lifetime of the Flagship project. These are as follows:

- the construction of the system software
- the evolution of the software towards the target system
- the interface with the host
- the persistent object store
- the interface with the hardware, and
- the performance of the system.

These issues are expanded below.

(i) *The construction of the system software*

The system software for Flagship will be implemented mainly in a functional language. This presents the problem of representing the state of the system and of composing the code implementing the different components of the system's state. Some work has been done on building operating systems using stream functions, with parts of the system composed using a non-deterministic merge[5]. This suffers from a number of disadvantages. Particular problems are the difficulty in handling dynamic systems where processes are created and deleted dynamically, the difficulty of handling complex scheduling of the processing of interactions and the necessity of manipulating explicitly the tags on messages in the streams.

Another method of handling the state of the system is that of *guardians*. This technique builds a stream processing function from a state transition function, and then hides the stream from the interface seen by users of the interface. This allows the software to be written as a set of state transition functions which are then converted to the required form by a "make-guardian" function. Guardians have been described in Denis[7], and are closely related to the concept of *managers* described in Arvind and Brock[6]. It is proposed that guardians are used as the basis of Flagship's system software.

(ii) *Evolution of the system*

The need to allow for major evolutionary changes in the system software is an extreme case of the usual problem of large scale software development. The techniques which will be used to alleviate the problems arising from the change will be simply those of good software engineering. The layered architecture will provide a stable framework in which change can take place. In addition the layer interfaces will be the subject of the main modelling exercises carried out during the development of the system, hopefully giving confidence in the design of the layers. The techniques of modularisation and "information hiding" will be used. The hardware dependent parts of the system will be isolated to a small kernel of the system. Also it will be an objective of the design of the system to build as much as possible using a set of replaceable components. The main technique for doing this will be to place as much as possible of the functionality above the CMI. This will have the additional advantage that such functions will be able to take advantage of the power of the CMI. Lastly, an important factor in allowing the easy

movement of functionality from the host to the Flagship machine will be the choice of the interface between the host and Flagship. This is discussed below.

(iii) *The interface with the host*

There are two main styles of interface which could be used between the host and Flagship: a message passing style, and a remote procedure call style. The latter will be used for Flagship. The reason for choosing to use remote procedure call mechanism is that it is very close to the style of interface used in both the host and in Flagship, making the movement of functionality from the host to Flagship very much easier. In addition it will be relatively simple to develop aids for automating the process of transition.

(iv) *The persistent object store*

The major cost in providing the CMI will be the provision of the persistent object store, especially as the construction of a persistent object store is a major research project in itself. Fortunately there is another Alvey project called the PISA project which is investigating just this area, and which will be implementing a persistent object store with the characteristics required by the Flagship project. This persistent object store will be used to provide the persistent object store of the CMI.

(v) *The interface with the hardware*

This is one of the main areas of research in the development of the system software. Particularly important areas of investigation are as follows:

Process scheduling. The problem of scheduling the work amongst the processing elements is clearly central to the success of Flagship. Hardware support is being developed for the distribution of work between the processing elements and for controlling the growth of parallelism in the system. However mechanisms for scheduling at higher levels need to be developed in order to be able to support user level facilities such as "break-in", "quit", and "continue".

Store management. There are many aspects of store management that are currently being investigated. Firstly the question of whether or not a virtual store mechanism should be included in Flagship is being addressed. Other issues include how to limit the amount of store a process can consume, and how the responsibilities of store management should be split between the hardware and the software.

Protection. An important area being investigated is the provision of some form of protection. This is necessary both to support secure multi-user processing and for the containment of the effects of errors.

Support for persistent objects. The hardware will need to provide support for persistent objects in a number of ways. For example the mechanism used for copying packets must avoid copying packets

representing objects (it is permissible to copy the value of an object but not the object itself). Similarly the hardware garbage collection mechanism must avoid garbaging objects but should inform the system software so that the object can be copied to stable store if necessary.

Performance. One of the main requirements on the system software is to provide feedback into the design of Flagship and its successors. Particularly important areas are the performance of the languages, the granularity of parallelism and the amount of concurrency in the system. Also very important is the performance of the load distribution mechanisms.

Having examined some of the issues arising in the design of the system software, it is appropriate to give a brief description of the first Flagship system. The main objective of the initial system will be to validate the main architectural concepts of the system, and in particular the hardware interface and the CMI. There will be two main steps towards this goal. The first step will be a pilot system which allows applications to be run on the Flagship hardware. Only a small subset of the CMI will be supported at this stage, in particular the persistent object store may not be provided. This system will allow investigations into the performance of the hardware to be carried out and to test the basic design of the system software.

The second system will provide full support of the CMI, including the persistent object store. This system is illustrated in Fig. 2. The diagram shows an application in Flagship being called from the host via the Remote Procedure Call mechanism. The application will be loaded into Flagship by the loader in Flagship accessing the persistent object store on the host – probably the Persistent Object Management System to be developed by PISA project (PISA POMS). The application will have access to the persistent object store, the host's operating system interfaces as well as the Environment Library in Flagship.

5 Implementation language issues

The Flagship system software will be implemented using several programming languages. The main one will be a high-level declarative language. At a lower level, DACTL will be used to implement efficient state-manipulating functions. At the lowest level, a machine code assembler is available for achieving effects not possible at higher levels. Although the strategy is to use the highest level wherever possible, it will be necessary to use lower levels and this cannot be ignored.

Initially the high-level language will be Hope+ (pronounced Hope plus), which is based on the research language Hope[16], with some extensions to provide for large-scale programming activities. In the longer term this will be replaced by a new declarative language, specifically designed to exploit the Flagship architecture.

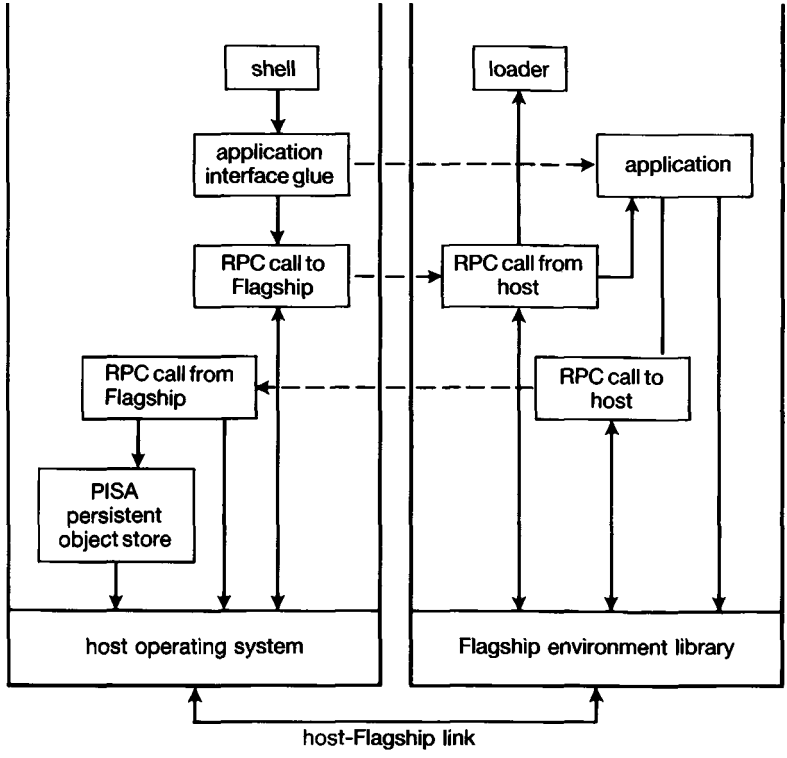


Fig. 2 Initial Flagship system

The extra features of Hope+ can be summarised as follows:

- The basic type Num has been extended to include negative integers.
- A basic type Real has been added to support positive and negative real numbers.
- An 8-bit ISO character set has been adopted as the internal character set.
- A standard module to handle Vectors has been included.
- Hope modules are supported, to allow separate compilation.
- Lazy evaluation is introduced for all data constructors. Function arguments are evaluated eagerly until a constructor exists at the outermost level.
- Partial parameterisation of function calls is supported to facilitate work with higher-order functions (this is a generalised form of "currying").
- Local mutual recursion is supported.
- Rule selection using pattern matching has been enhanced to permit overlapping patterns, resolution is based on a tightest-fit strategy.
- Exception handling is supported by means of a special value "error" which is a member of every type's value space.

The implementation model presented so far is fairly straightforward, since the languages considered are closely related; Hope+ compiles into DACTL

and DACTL compiles into the Flagship machine code. However the situation is complicated by the need to develop the system software on host machines. This requires inter-working between the "Flagship" languages and the "Host" native languages (S3 for the VME host and C for the UNIX host). A further complication arises concerning the persistent object store; initially this will be provided by a system developed by the PISA project which offers PS-Algol and the PISA Persistent Object Management System (POMS). This system will run on the host and act as the stable store for Flagship, so inter-working with PS-Algol will be necessary. The detailed requirements are being evolved, but a good deal of work is necessary to ensure that all the required combinations of language inter-working are possible.

6 Conclusion

Some important themes have recurred throughout this paper; themes fundamental to our task:

- We are designing our system to meet the challenge and complexity of 1990s applications.
- We are basing much of our implementation on the use of existing (UNIX and VME) operating system functions to allow us to concentrate on the research and development of the declarative system.

Our task is a difficult one with so much that is novel and unknown, but the PRM, CMI and other system characteristics are being very carefully specified to provide a sound underlying design. We are adopting design methods such as the use of formal specification and functional languages. All of this aims to give a quality system which meets the needs of its users, supplies the demands of the 90s, and is right first time.

References

1 WATSON, I.L., SARGEANT, J., WATSON, P. and WOODS, V.: 'Flagship computational models and machine architecture'. ICL Technical Journal 5(3), 555–576, May 1987.
2 TOWNSEND, P.: 'Flagship hardware and implementation'. ICL Technical Journal 5(3), 575–594, May 1987.
3 DARLINGTON, J.: 'Software development using functional programming languages'. ICL Technical Journal 5(3), 492–508, May 1987.
4 JONES, C.B.: 'Systematic Software Development Using VDM'. Prentice Hall International, 1986.
5 JONES, Simon B.: 'A Range of Operating Systems Written in a Purely Functional Style'. Programming Research Group Technical Monograph PRG-42, Oxford University, September 1984.
6 ARVIND and BROCK, J.D.: 'Streams and managers'. Proc. 14th IBM Comp. Sci. Symp., Lecture Notes in Comp. Sci., Springer-Verlag, New York, 1982.
7 DENIS, J.B.: 'Data should not change: a model for a computer system'. Lab. for Comp. Sci. CSG Memo. 209, MIT, Cambridge, Mass., 1981.
8 ATKINSON, M.P., BAILEY, P., COCKSHOTT, W.P., CHISHOLM, K.J. and MORRISON, R.: 'Progress with Persistent Programming'. Technical Report PPR-8-84, University of Edinburgh, 1984.
9 CARDELLI, L. and WEGNER, P.: 'On Understanding Types, Data Abstraction, and

Polymorphism'. Technical Report CS-85-14, Brown University, Providence, Rhode Island, August 1985.
10 CARTWRIGHT, R.: 'Types as Intervals'. ACM 0-89791-147-4/85/0001, 22–36.
11 CARDELLI, L.: 'A Semantics of Multiple Inheritance'. In: Semantics of Data Types, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.
12 LISKOV, B.: 'Overview of the ARGUS Language and System'. MIT Programming Methodology Group Memo 40, Feb. 1984.
13 KRABLIN, G.L.: 'Building Flexible Multilevel Transactions in a Distributed Persistent Environment'. In: Persistence and Data Types Papers for the Appin Workshop, PPRR 16, University of Glasgow, August 1985.
14 BRENNER, J.B.: 'A General Model for Integrity Control'. ICL Technical Journal. 1(1) 71–79, November 1978.
15 'Department of Defense Trusted Computer System Evaluation Criteria'. CSC-STD-001-83, Department of Defense Computer Security Center, Maryland, August 1983.
16 BURSTALL, R.M., MacQUEEN, D.B. and SANNELLA, D.T.: 'HOPE: An Experimental Applicative Language'. Dept. of Computer Sc., University of Edinburgh (CSR 82).

Flagship computational models and machine architecture

Ian Watson, John Sargeant, Paul Watson and Viv Woods
Dept. Computer Science, University of Manchester, Manchester M13 9PL, England

Abstract

The Flagship project aims to produce a computing technology based on the declarative style of programming. One major component of that technology is the design for a parallel machine whose computing power can be increased simply by the addition of hardware resources. This paper describes the principles of such a computer architecture.

1 The case for parallel computer structures

Many of the advances in computer design during the last decade have been a result of the increase in scale and decrease in price of integrated circuit components; particularly the development of CMOS VLSI. The major impact of this has been in the personal workstation sector of the market where machines, with computational performance previously associated only with mainframes, are available at a very low price.

If one combines special high performance technology with sophisticated design it is possible to achieve a performance level a few times greater than a cheap workstation. However, it is becoming clear that the rate of increase of performance of cheap VLSI circuits is significantly greater than that of higher performance technology.

Networks of workstations are able to share the resources of distributed filestore systems, providing a single user with local computing power, but access to a global information system. The problem arises when an individual's computational requirements are greater than can be provided by his local workstation. A high performance mainframe can satisfy occasional demands for individual peaks up to its performance limit, current distributed network systems cannot harness the combined power of their individual components.

In many cases the computing performance currently utilised by an individual user is limited more by the resources available than his ability to make sensible use of anything greater. As information processing increases in

complexity, particularly with the advent of knowledge based systems, users will require performance greater than can be provided even by current mainframes.

Given this set of conditions, there is an obvious solution. It is simply necessary to find a way of utilising an arbitrarily large number of cheap VLSI processors which can act as a homogeneous computing engine. The resulting resources can be shared or pooled as necessary and the system can be extended arbitrarily if the total performance is inadequate. The solution is so obvious that, with the exception of a few special purpose parallel structures suited mainly to the solution of particular types of scientific problem, it is astounding that after forty years of computer development we are still using single processors executing serial sequences of instructions.

It is not difficult to envisage such a parallel structure. Clearly, if the processors are going to co-operate on the solution of problems, they are going to need to communicate. We can guess that they may need to communicate quite often and thus we need to provide high bandwidth communication channels between them; but given current technology this is not really a problem, we can provide sufficient capability to communicate every few instructions. So the hardware is easy, what about programming? All we need to do is to extend our languages so that we can specify when and where communication takes place.

The first major experiments in this area were conducted over twenty years ago. They have been repeated many times since with very similar results; it has proved impossible to exploit the hardware potential of the machine beyond a very small number of parallel processors.

The problem is mainly one of dynamic variation. Communication in any real program varies both in time and place, probably as a function of particular sets of data. It is not, in general, possible to perform a static partitioning of a program into separate communicating serial sections in a manner which will utilise the machine resources efficiently. To achieve efficiency it is necessary to make use of both dynamic allocation of resources and dynamic division of a problem into appropriately sized parallel sections.

The communicating process approach found in languages such as Ada and Occam is unable to provide such dynamic flexibility. Apart from this there is also an issue of software complexity. The writing of a conventional program requires one to keep track of the order and position of data produced and utilised by the program. Failure to manage this complexity is the major contributing factor to software error. If one tries to introduce the additional time variance of data production in parallel program sections, this situation can only get worse. Given the major concern which exists worldwide over the correctness, reliability and safety of computer software, it would seem unwise to move in this direction.

2 Declarative languages and parallelism

Current computing languages have developed alongside serial computing machines. As a result they consist of sets of serially executed instructions which specify how data is operated upon and moved between storage locations; in fact just a higher level encoding of the basic machine order code.

In order to enable dynamic division of a problem into appropriately sized parallel sections it is desirable that all parallelism which exists is available in the problem specification. Clearly, conventional languages with their inherent serial structure do not have this property. However, if the specification of parallelism adds to the complexity of programming it would be a mistake to require every little bit to be expressed.

Fortunately there is a way out of this dilemma. The requirement that the parallelism is available in the program does not imply that it must be expressed explicitly. The answer lies in simple mathematical expressions. Given something of the form

$$a = (b + c) * (d + e)$$

it is clear that the two additions can be performed in parallel (or performed serially in any order) followed by the multiplication. We did not need to say that there was a process which added b to c and one which added d to e and that they then both communicated their results to a process which multiplied them together. The parallel structure is all there but it is implicit.

In general, we know from simple mathematics that, given any arbitrary expression, the whole evaluation process is simply one of evaluating sub-expressions repeatedly until no further simplification is possible. Remembering the example above, it should be clear that large expressions may have significant amounts of parallelism in the evaluation of their sub-expressions. If we can devise a computing machine which works wholly on the basis of simple mathematical expression evaluation then we may be able to achieve the desired characteristics.

The next question is clearly "can mathematical expressions constitute computer programs?" A detailed answer to that question is outside the scope of this paper; the simple answer is "yes", with some reservations. Clearly there is a similarity between conventional programs and simple expressions, we could have written the above example directly as a FORTRAN instruction. The major differences are:

1 In a computer programming language, we are allowed to re-use variable names in a way which does not make sense mathematically (e.g. $x = x + 1$). However, this is (mostly) just a convenience to enable us to re-use storage locations. If one prohibits the use of variable names more

than once on the left hand side of expressions then all simple assignments become equivalent to expressions.

2 Conditional computation. Various control structures are available in a programming language to modify the progress of the computation as a result of data comparisons. All real computation makes use of this facility. Fortunately, it is possible to introduce a conditional expression into our mathematical notation to provide equivalent power.

3 Data structures. Simple expressions deal only with simple values such as integers. Real programs make heavy use of facilities which enable us to group data in various ways to form larger units which can be further manipulated. However, it should be clear that such structuring is also used in many branches of mathematics; for example, expressions containing vectors, matrices, etc.

Programming languages based on simple mathematical expressions are therefore possible. Indeed many believe that they are positively desirable for reasons other than those already outlined. They are amenable to formal reasoning about their function and are particularly suited to the expression of highly complex computations.

The detailed structure and use of declarative languages are covered elsewhere in this issue[1-4]. The remainder of this paper will concentrate on parallel machine structures for their execution based on the principle of packet based graph reduction which are being developed as part of the Flagship project. Much of the early work in this area was pioneered by the Imperial College ALICE project[5].

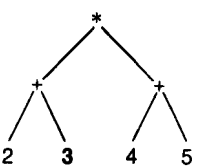
3 Graph reduction

The program is a mathematical expression and the process of computation is one of evaluating sub-expressions (in parallel where appropriate) until no further evaluation is possible. If we are to construct a machine to perform this process then it is necessary to consider how expressions can be represented in a practical form.

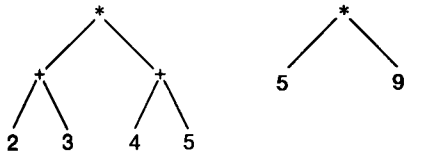
On paper they are written as serial strings of characters, but they have additional structure which is best represented graphically. For example, the expression:

$$(2 + 3) * (4 + 5)$$

can be shown as:



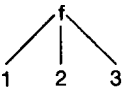
This brings out both the sub-expression structure and the parallelism. The process of evaluation is to perform the operations at the leaves of the tree thus simplifying the graph; this process is usually referred to as reduction. The stages of the process would be:



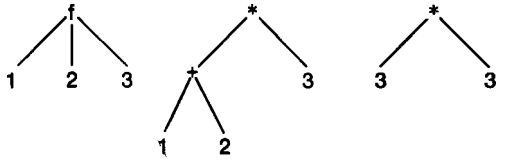
So far we have considered expressions only of arithmetic operations. These are particular cases of functions which have fixed reduction rules. In general, expressions will consist of user defined functions where the reduction rules for the function constitute the program. For example, given the definition:

$$f(x,y,z) = (x + y) * z$$

We might start with the graph:



and proceed with the reductions:



The process is therefore one of applying functions to arguments according to a set of reduction rules. A practical machine would regard these operations as the fundamental units of computation and an appropriate physical representation of the graph must be chosen. The computational graph is therefore represented as "packets", where a packet is a function together with the arguments to which it is applied. The above example is then:

| f | 1 | 2 | 3 |

which becomes:

| * | ^ | 3 |
\
| + | 1 | 2 |

then

| * | 3 | 3 |

and finally

| I | 9 |

the "I" being an integer "constructor function" indicating a piece of data in minimal form. These packets are assumed to exist in a conventionally addressable store where the links between them forming the graph are pointers to the appropriate packets. Each field of a packet will, in general, need to hold an amount of information equivalent to a word in a conventional machine; a packet is therefore a collection of words in adjacent store locations. Parallelism can be exploited by choosing any packets which are in a form to be reduced and performing that reduction in multiple processors.

4 Abstract machine architecture

A physical machine might have an abstract structure similar to that shown in Fig. 1.

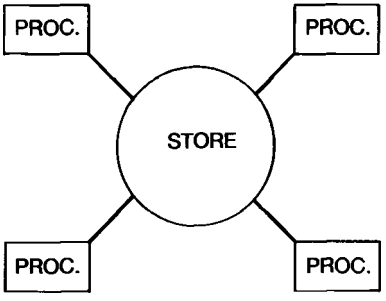


Fig. 1 Abstract machine structure

The processors access the store, select a packet, perform a reduction and place any resulting packets back into the packet store. The underlying principle of operation is hence very simple; however there are a number of detailed issues which need further explanation. In the following sections we will assume that the processors contain definitions of the functions and hence know how to perform reductions for any packet which they may encounter. The exact form of these definitions will be examined in a later section.

4.1 Packet modes

In general, only a subset of packets in the graph will be candidates for reduction at a particular point in time. Some will be waiting for the sub-

expressions representing their arguments to reach a form where reduction is possible. Others may already be in a state where no further reduction is appropriate. Although it may be possible to derive the state of a packet by examination of its contents it is clearly sensible to tag the packets with a "mode" which indicates the current state. We will introduce three such modes.

! Active Mode. The packet is a candidate for further reduction.
#n Suspended Mode. The packet is waiting for n of its arguments to reach a form where reduction is possible.
– Dormant Mode. The packet is not currently a candidate for any action.

The practical use of these is best illustrated by example. Given a function definition of the form

$$f(x,y) = (x + y) * (x - y)$$

and a call to that function:

f(4,2)

we would start with a packet in ACTIVE mode:

@a! | f | 4 | 2 |

This would be processed according to the function definition to produce:

@a#2 | * | | |

@b! | + | 4 | 2 | ^a.1 |

@c! | – | 4 | 2 | ^a.2 |

Note that at this point we have introduced packet addresses into the picture using the @n notation. It is also necessary to introduce an extra packet field which indicates where the value of a function is to be returned when it is computed. In this case the add and subtract operations will return integer values back to the argument fields of the multiply, shown using the notation ^n.m i.e. a pointer to the packet whose address is n together with the field number m. The evaluation of the two active packets will produce:

@a! | * | 6 | 2 |

and finally:

@a~ | I | 12 |

The top packet of the graph is shown without a return address for its answer.

In this simple example this is an integer representing the complete answer of the computation and we have simply overwritten the packet with an integer constructor function which needs no further processing. In a practical computing system the answer would probably be communicated via a system "procedure" to a printer, file, etc.; the top packet would thus contain the address of such a procedure.

4.2 "Eager" and "demand driven" evaluation

In the above example, we assumed that the processing of the first active packet produced two further active ones, the add and the subtract, and a suspended multiply waiting for their results. Information concerning this must form part of the function definition code. The packet modes provide for different evaluation schemes which can prove valuable in the implementation of declarative languages. For example given the same function as before, from the packet:

@a! | f | 4 | 2 |

we could have produced:

@a! | * | ^b | ^c |

@b~ | + | 4 | 2 |

@c~ | − | 4 | 2 |

In this case the next stage of the processing would require the reduction of the multiply operation, but this is not possible because its arguments are not in integer form. We must assume that the rules which the processor contains for the reduction of the multiply function know of this requirement. They therefore turn the multiply into a suspended form, activate the packets which are its arguments and attach the return pointers.

@a#2 | * | | |

@b! | + | 4 | 2 | ^a.1 |

@c! | − | 4 | 2 | ^a.2 |

We now have proceeded to the same stage in the evaluation that we reached during the previous example after the first reduction. This may appear as simply an inefficient evaluation scheme, particularly for such simple integer computation. However, it illustrates the principle of "demand driven" or "lazy" evaluation, namely that we do not evaluate the arguments to a function before the call takes place, but wait until the function evaluation itself demands that evaluation.

This principle has important uses in the declarative programming style, particularly in the handling of data structures. The expressive powers of the languages are greatly enhanced if finite data structure sizes do not need to be specified. In fact, we allow the expression of infinite data structures but obviously do not evaluate elements of such an infinite object until they are needed at run time. The simple mechanism described here is sufficient to permit the implementation of all such features.

4.3 Data structures as graphs

A data structure is simply a collection of elements which can be viewed as itself an element. Although data structures appear in many forms in programming languages there are really only two fundamental concepts involved in their construction.

The first is that of locality; given the structure as a whole it is possible (usually by some simple indexing mechanism) to access any element. Arrays and records are the most common example of this form of structure, although the tuple should probably be regarded as the canonical form. It should be clear that the concept of a packet described previously as just a collection of related addresses in a physical store is itself a tuple data structure and, given such a packet structure, the locality aspect of data structures should present no conceptual problem. It does suggest, however, that our packets may need to come in a wide variety of sizes.

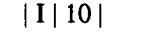
The second property of data structures is a result of the recursiveness of their definition; any element of a structure may itself be a structure. This is already familiar from conventional programming languages which allow arrays of arrays, etc. although there may often be particular implementation restrictions.

In declarative languages, full recursiveness is allowed in all definitions. The most common form of declarative language structure is the list, which contains both principles described. It is a two position tuple, the first position containing an element and the second containing a list. In order to implement such structures we simply permit any element of a tuple to be a pointer to another tuple. It should be clear that such a structure is simply a graph of packets.

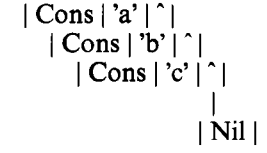
We have already mentioned the concept of an integer constructor function as one which is simply a value which has no further reduction rules. It is in fact a particular example of a data structure which contains a single element. We would normally associate a constructor function name with any data structure tuple. In general, a constructor function has no reduction rules and therefore any graph containing only packets with constructor function names is a stable evaluated structure.

Examples of structures are:

I(10)

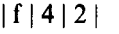


Cons('a',Cons('b',Cons('c',nil)))



[A,B,C,D,E]

It should be noted that if we want to maintain consistency in our treatment of data structures then packets of the form:



should strictly be represented as:

However, in any practical system, it clearly makes sense to have a representation for a set of common atomic values which can be embedded in place of a pointer to their full form. Packet processing must always be easier, the less information which needs to be obtained via separate store accesses.

4.4 Sharing – more complex graph structures

All the physical representations which we have dealt with so far are strictly trees rather than graphs. That is they have sub-structures which are entirely independent. However, consider an example of the form:

$$f(s) = g(s) + h(s)$$

where the argument s was passed to the function f as a pointer rather than an embedded atomic value. This could happen either because s was some graphical data structure or we were using demand driven evaluation and it was, as yet, an unevaluated graph. We would therefore start with a graph of the form:

@a! | f | ^s |

@s~| etc | |

The next step in the reduction requires the argument graph to be referenced twice from the newly created packets. There are two implementation choices, we can either make complete extra copies of the argument graph where necessary or we can simply copy the pointer.

The first solution is clearly inelegant and inefficient. If the graph is a structure, then we may need to copy very large amounts of information although we will not be able to determine how much of the copied graph will be required for subsequent computation. If it is an unevaluated sub-expression we risk multiple re-evaluation. If such an unevaluated graph is copied through several stages of function call then the potential for re-evaluation will grow exponentially.

Copying of the pointer alone is simple and elegant, it is in fact this scheme which is correctly termed graph reduction. Its major drawback is that it destroys the simple tree structure of the computation. Now any packet in the store may be multiply referenced via pointers from other packets. This has several consequences:

(a) In our simple examples, an active packet had a single field in which it held an address to which it would return its value. Now an active packet may need to hold a large number of such addresses. Moreover, requests to activate a packet can originate from multiple sources and hence any particular request may encounter a packet in a variety of modes, we need nechanisms to cope with this.
(b) In the simple examples, we assumed that a packet which became an integer value simply returned that integer as an embedded field and the packet then ceased to exist. Now we must overwrite the packet with an integer constructor in addition to the returning of a value because it may be shared by other parts of the computation which have yet to reference it.
(c) Given that we preserve all computed packets in case they are referenced further we introduce a storage management problem. We need to introduce "garbage collection" mechanisms to reclaim storage when it is certain that it will no longer be referenced. Such mechanisms can become complex particularly in parallel systems.
(d) Our simple abstract machine picture uses a central globally accessible store. It is clear that a practical machine structure will almost certainly require this store to be parallel and distributed from considerations of storage bandwidth. If we wish to maintain flexibility in the distribution of our graphical computation across such a store it is clear that all store must be globally addressable.

Of these, the need for garbage collection and globally addressable memory are the most serious, particularly in parallel computer systems.

The scheme of copying all arguments is generally known as "string reduction" as it is equivalent to the evaluation which results from the naive implementation of expression evaluation using textual strings. Its most attractive feature is that it maintains the simple tree structure of the computation in that all sub-expressions are independent. This allows the spreading of sub-computations across parallel machine structures in a simple way and without the need for globally addressable memory space. For this reason there have been several proposals for parallel machines based on string reduction or schemes which are largely equivalent[6]. Although there may be certain classes of problems to which it is suited, it is widely believed that, for most general purpose computing, the cost of copying and re-evaluation will far outweigh any advantages to be gained from string reduction.

It should be emphasised that graph reduction is the more general case and that string reduction can readily be implemented in any graph reduction machine structure in those circumstances where it may be worthwhile.

5 Practical machine structure

The probable need for distributed parallel store has already been mentioned. An extensible machine structure must allow parallelism in both processing and storage if total computing power is to be increased by the addition of extra hardware. It is clear that a physical realisation based directly on the abstract picture would soon reach a performance limit as processors were added, owing to the inability of the shared store to cope with the rate of access. Indeed, if one is considering 'cheap' VLSI technology, the performance of a single processor is often limited by storage bandwidth; in these circumstances parallel stores are essential to any multi-computer structure.

5.1 A remote store-processor structure

It must be emphasised that all such storage needs to be accessible from any processor in the system if we are to have a flexible implementation of full graph reduction. The simplest way to achieve this is by using a physical structure like that shown in Fig. 2.

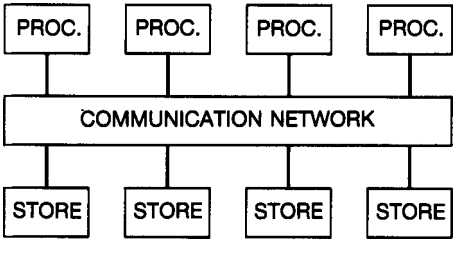


Fig. 2 A simple distributed machine structure

All stores are accessible across the communication network from any processor. If we can organise the distribution of the computation in such a way that the store-processor access patterns are favourable, then the full parallel bandwidth of the stores should be available. In practice the easiest way to achieve a favourable distribution is by randomising the packet allocation across the stores.

In a conventional serial computer the store processor interface often has a critical effect on the machine performance. Many of the common machine architectural principles such as internal registers, caches, instruction pre-fetching, pipelines, etc. have been developed in order to overcome store processor bandwidth limitations. In the structure shown above the interface includes a communication network which, however it is implemented, will suffer a degree of latency, contention delays, etc. One of the major advantages claimed for the graph reduction model of computation is that communication delays are relatively unimportant because, owing to the lack of ordering required for sub-expression evaluation, a processor can proceed with other work while data is being fetched. Practical experiments such as the ALICE prototype have indicated that, although this proved true to a limited extent, at least in comparison with more conventional multi-processors, the sheer loss of total bandwidth resulting from the remoteness of the coupling is a serious impediment to the performance of each processor in the system. One major argument used in favour of parallel systems concerns the cost-performance of cheap VLSI technology. If one has to sacrifice a significant factor in the performance of each individual component in order to produce a total workable system, then many of those arguments are negated.

5.2 A closely coupled store-processor structure

It is proposed that a Flagship machine will employ a closely coupled processor-store structure as shown in Fig. 3.

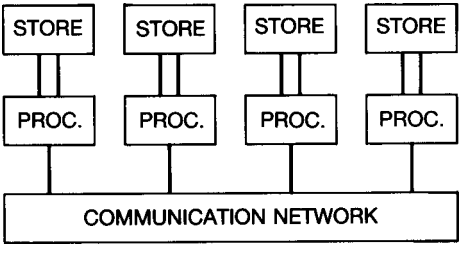


Fig. 3 A closely coupled structure

Using this form of structure, the intention is that the majority of store processor interactions will be between a processor and the local store to which it is closely connected. In general therefore, a processor will attempt to process only those packets which exist in that local store. In previous

descriptions of the computational process we have referred to a processor fetching a packet and producing new packets, considering a packet operation as the manipulation of a complete unit. In practice many packet operations require only modification of the contents of part of the packet. In a closely coupled structure, it is considerably easier to organise such partial modification with additional saving in store bandwidth requirement.

Remembering the requirement for global addressability of the complete graph, we must now consider how the computational graph can be mapped onto this structure and what happens when store accesses are not local. In the remote storage structure, the computational graph can be spread randomly across the parallel stores, indeed this is often desirable to minimise network contention. However, in a closely coupled structure the requirement is reversed. Clearly, the intention is to ensure that as many accesses as possible are to local store during the computation process.

If a processor only takes active packets from its own store then these accesses are always local. However other accesses take place during the reduction process if a packet has pointers to other packets which form its arguments. This may be because they are data structures or unevaluated pieces of graph. What we want to achieve is a partitioning of the graph in the stores so that the majority of argument accesses are local; however it is inevitable that some pointers will cross store boundaries and non-local accesses will occur. We need to examine how a favourable situation can be achieved.

5.3 Mapping the computational graph

The obvious strategy is to maximise the number of pointers which are internal to a particular store. The best way to achieve this is to map the graph across stores in such a way that locally connected clusters of packets are kept together; this is indicated pictorially in Fig. 4.

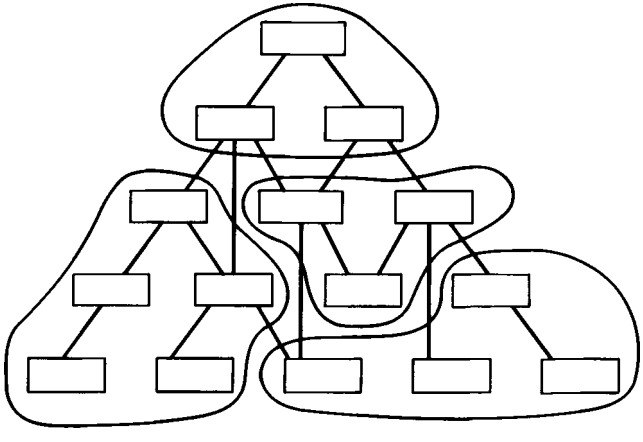


Fig. 4 Clustered graph mapping

In general it is to be expected that any one store will contain more than one such cluster, although the aim should be to achieve a small number of large ones.

We have already mentioned the difficulties which arise with static mappings of parallel computation onto parallel machines. We therefore require the boundaries of our graph mapping to be dynamic, being adjusted according to the utilisation of resources in the system. For example, if a particular processor runs out of active packets of its own to process, it will signal to other processors which, if they have an excess of work, will export active packets. The details of this mechanism are beyond the scope of this paper, but can be found in work published by Sargeant[7].

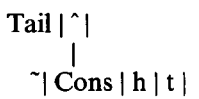
One crucial issue in the operation of such a scheme is how the packet groupings are achieved, and whether they can be maintained during the execution of the computation. The nature of declarative programs is that they usually start with a single active function call and the reduction process produces an expanding graph which will reach a maximum size and then contract. In this regime it is clear that the spreading of the graph across the stores must be done almost wholly at run time. The positive aspect of the computation style is that the graph tends to expand and contract around local centres rather than undergo total random re-connection; it is hoped that this will tend to maintain local clusterings once they are established. When a processor is required to export work to others, which is clearly the process which achieves the graph distribution, the obvious strategy is to export those packets which are already at the edges of a cluster rather than in the centre. This whole aspect of the machine operation is one which can only be evaluated by observing the dynamics of real programs, as is currently being done by simulation.

5.4 Access across local boundaries

If, during the process of a packet reduction, the processor requires to know the contents of an argument packet to perform that reduction then that argument must be available in the store to which the processor is closely connected. This situation occurs in the reduction of functions which have data structures as arguments, for example a call to the function:

Tail(Cons(h,t)) = t

would appear in packet form as:



and the processor would need to access both the active "Tail" and the "Cons" in order to perform the reduction. If the "Cons" was not local in the store, it is

necessary to make a local copy. Such data will sometimes be shared by other parts of the computation in that same processor and it would be advantageous if any copy made could be shared locally. This is achieved using a system of distributed virtual memory. The local stores of the processors are regarded as real instances of areas of a global virtual memory space. Further details of the mechanisms involved can be found elsewhere[8].

6 Specifying the reduction process

6.1 Form of the code

The instruction code of a graph reduction machine is the information obtained from the program function definitions which specifies how a packet is to be reduced; new packets to be produced, their contents and the modes in which they are to be created. This information is then used by a single processor. As such, it is held in the form of serially executed instructions which specify general packet manipulations. It is not unlike conventional machine code except that it contains instructions which are optimised for the operations involved in packet reduction.

We have previously used simple arithmetic examples to indicate the operation of packet reduction, for example the function:

$$f(x,y) = (x + y) * (x - y)$$

and a call to that function:

$$f(4,2)$$

starting with the packet

@a! | f | 4 | 2 |

and producing the set of packets:

@a#2 | * | | |

@b! | + | 4 | 2 | ^a.1 |

@c! | − | 4 | 2 | ^a.2 |

In a practical machine this would in most cases not be a particularly efficient thing to do. If the reduction rules are held as relatively conventional code then it would be possible to perform the complete function evaluation in registers rather than create a complete computational graph for such a simple expression. This will produce a significant saving in store processor communication and hence more efficient run-time performance. This technique will however reduce the amount of parallelism which is able to be

exploited in simple expression evaluation and experience with practical systems is needed to determine suitable compromises.

It should be noted that such optimisation can only be performed if the function is called with its arguments ready evaluated. Lazy evaluation is therefore likely to introduce inefficiency and should only be used where absolutely necessary.

6.2 Code in packet form

Although the code is a set of serially executed instructions, it is divided into sections, one for each function reduction required for the computation. The appropriate piece of code to apply to a particular reduction is determined by the "function name" in the first field of the packet to be reduced. In practice that function name will be a pointer to the start of the appropriate instruction block held in the store of the processor.

One possibility is to hold all function definitions in all processors. This would, however, be wasteful of store. It also leads to serious organisational problems when one considers aspects such as compilation and code loading. In these circumstances it is necessary to treat code very much like data. If, instead, the sections of function code are themselves held in packets and viewed at the machine level just as other pieces of data, then a number of distinct advantages emerge.

The writing of the sorts of system programs mentioned above is greatly simplified. In addition if one introduces the concept of laziness in the construction of "code data packets" using lazy evaluation mechanisms which already exist, it becomes possible to produce, very easily, incremental compilation systems and those sorts of programs which add to their own program structure as part of their run-time behaviour. Such techniques, although not always yet fully understood in a declarative programming world, appear essential to the production of intelligent computing systems of the future. One final advantage which is relatively mundane but nevertheless important is the removal of the need for all processors to contain all code. The appropriate data structures can now be copied around the machine on demand using exactly the same mechanisms as those provided for the data.

7 Other issues

There are a number of other issues which are being investigated as part of the Flagship machine architecture project. The detail is beyond the scope of an introductory paper, but the areas deserve brief mention.

7.1 Variable size packets

It is apparent from the descriptions of various aspects of the machine structure that a packet of information may contain anything from a very

simple machine operation with a couple of arguments through to data structures which may be of significant size.

In practice, one can always construct graphs which behave in the appropriate manner from packets of a small size. For example, the necessary random access property of an array can be achieved by building a tree together with appropriate access functions. Normally there is a penalty of efficiency to be paid for such a technique. On the other hand the provision of arbitrarily sized packets of information introduces complexity, particularly in the management of storage allocation.

The ALICE machine used only one size of packet with five fields. Although we do not want to go to the full complexity of totally variable sizes, we feel that there are significant benefits to be gained from providing a limited set of packets ranging from a few fields up to several hundred. We believe that this will enable the implementation of certain necessary forms of data, e.g. arrays, without significant loss of efficiency, whilst avoiding undue store management complexity. It is worth pointing out here that the allocation of huge contiguous areas of store is inconsistent with the need to distribute computation and data structures across any parallel machine.

7.2 Garbage collection

The need to reclaim storage areas is a consequence of the graph reduction model of computation with the sharing of graph structures. Because the rate of usage of store is high, it is necessary to perform storage reclamation automatically rather than leave it to the user as, for example, in heap management systems in PASCAL. The problems and techniques of automatic storage reclamation have been studied for many years, particularly in connection with the LISP language which is closely related to the declarative languages which Flagship machines will support.

Unfortunately, many of the known techniques are not well suited to implementation on parallel machines. The most widely used LISP mechanisms, known as "mark-scan", "stop and copy", etc., require that the computation is stopped and the machine resources devoted to working out which sections of the computational graph can be reached from all useful parts of the computation. These are preserved while all other storage is returned to a pool which can be re-allocated. Such mechanisms are possible in a parallel structure but more complex and probably undesirable.

Another technique known as "reference counting" is more suited to parallel machines. Here, a count of the number of pointers which exist to a particular packet is maintained as part of the packet structure. This count is continually adjusted as pointers are created and destroyed and, if it reaches zero, the store is reclaimed. The major limitation with this method is that it is unable to deal with circular graphs in its simple form.

Two major areas of importance are the necessity of circular structures in a Flagship environment and the possibility of alternative or related schemes which are able to deal with them.

7.3 Virtual memory and distributed I/O

The mechanism for moving data around the machine has been described as distributed virtual memory. It is likely that in a real machine structure this needs to be combined with secondary storage systems. With parallel stores and machines it is probable that any physical secondary storage devices will need to be distributed also.

We should also remember that many projected applications of future "fifth generation" systems will, in addition to the requirement for enhanced computational power, also involve the manipulation of large amounts of data. It is likely that the requirement for input/output bandwidth to storage devices holding background data will increase as a direct linear function of computational power. In these circumstances it is inevitable that the input/output devices must be distributed throughout the parallel machine structure in an integrated manner. High performance computational machines or high performance database machines as separate entities will have limited applicability. This aspect of Flagship machine architecture requires further careful consideration.

8 Conclusions

Extensible, general purpose, parallel machine structures are likely to prove necessary in future computer designs both to use the potential of VLSI in a cost-effective way and to provide ultimate computing power which will be unobtainable from serial structures.

The graph reduction parallel model of computation provides a sound framework on which to build a parallel machine structure which does not suffer from many of the disadvantages of more conventional computing approaches. Declarative languages which lend themselves to such implementation techniques are arousing interest for separate reasons. They are claimed to be easier to use for many complex programming tasks and they have the advantage that their mathematical basis provides possibilities for formal program manipulation and proofs.

Prototype parallel machines in this area have verified the basic viability of the approach and provided a wealth of experience on which to base the design of future systems. One critical issue in the design of a real machine is the performance which results from computing resources in the system. Early prototypes have suffered significant performance penalties by departing significantly from conventional machine architectural principles.

Conventional multi-processors have failed to gain wide acceptance because

their parallel facilities proved very difficult to utilise in a conventional programming style. However, it is now believed that parallel machines built from closely coupled store/processor elements provide the most effective parallel structure and that these elements can achieve efficiency from executing relatively conventional code. That is not to say that those elements are just conventional machines. As in all modern computer design it is essential to provide support for particular aspects of the software system and languages.

There are a number of other issues which need to be addressed in the design of parallel machines of the future, probably the most important of which concerns the way in which the computing power of the structure can be utilised in conjunction with the large amounts of data which will inevitably need to be accessed.

The Flagship project is pursuing all these areas of parallel machine development and hopes to produce practical machine designs within the next few years.

References

1 BABB, E.: 'Language overview'. ICL Technical Journal 5(3), 471–476, May 1987.
2 ATKINSON, P., MORRISON, R. and PRATTEN, G.: 'PISA — a Persistent Information Space Architecture'. ICL Technical Journal 5(3), 477–491, May 1987.
3 DARLINGTON, J.: 'Software development using functional programming languages'. ICL Technical Journal 5(3), 492–508, May 1987.
4 GLAUERT, J.R.W., KENNAWAY, J.R. and SLEEP, M.R.: 'Dactl: a computational model and compiler target language based on graph reduction'. ICL Technical Journal 5(3), 509–537, May 1987.
5 DARLINGTON, J. and REEVE, M.: 'ALICE–A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages' Proc. 1981 ACM Conference on Functional and Computer Architecture.
6 BURTON, F. and SLEEP, R.: 'Executing Programs on a Virtual Tree of Processors'. Proc. ACM Conference on Functional Programming and Computer Architecture, 1981, 187–194.
7 SARGEANT J.: 'Load Balancing, Locality and Parallelism Control in Fine Grain Parallel Machines'. Internal Report, University of Manchester, Dept. of Computer Science, 1986.
8 WATSON, I. and WATSON, P.: 'Graph Reduction in a Parallel Virtual Memory Environment'. Proceedings of the MCC Graph Reduction Workshop, Santa Fe, New Mexico, 1986, Springer-Verlag.

Flagship hardware and implementation

Paul Townsend
ICL Mainframe Systems, West Gorton, Manchester

1 Introduction: Flagship machine objectives

The objective of the Flagship hardware development is to provide an extensible parallel machine for the early 1990s. Parallelism will allow higher performance than conventional machines, yet it will permit cost-effective technology to be used. This will ensure that performance is achieved more cheaply than the current conventional approach of using more and more complex and costly hardware technologies.

Declarative languages, both functional and logic, will be supported. The design philosophy is one of languages first, meaning that the machine is tailored to these languages to provide the attributes that the languages require. Cost/performance is paramount and there should be little conflict in efficiently providing the necessary language attributes and achieving high system performance.

The Flagship machine will be extensible over a range of numbers of processors. As the machine supports irregular* parallelism as well as regular there is no reason why it should have any special number of processors: 7 or 97 could be accommodated in the design. For physical and logic partitioning reasons we are likely to extend the machine by binary increments between 8 and 256 processors.

The Flagship machine will be packet based, its smallest discrete item being the packet. Sub-fields within a packet can only be accessed via the packet identifier. Graph reduction principles will be used[1]. A parallel distributed operating system will be supported, with many I/O channels operating in parallel.

Within the constraints of being declarative-language based Flagship will be

*Irregular parallelism. Many machines exploit only regular parallelism and therefore all processors at any one time are computing in a similar way: these are the Single Instruction Multiple Data (SIMD) machines. Matrix multiplication, for example, would suit such regular parallel computations. Irregular parallelism implies that though a program has many parallel threads that could be computed in parallel, these threads may be computationally very different in nature. Also the amounts of parallelism may change dynamically as the program progresses. This requires Multiple Instruction Multiple Data machines (MIMD), with many processors performing different tasks at one moment. Flagship is a MIMD machine.

"general purpose" and is not aimed at a narrow set of application domains. For highly parallel problems the machine should approach linear speed-up as more processors are added – meaning that if the number of processors is doubled then the performance should almost double.

2 Flagship development route

There are many major research issues which need to be addressed by the project; therefore the approach has been one of incremental development. There are four main stages, as below.

Stage 1

The first prototype developed by the project was ALICE. In the early 1980s a team led by John Darlington and Mike Reeve at Imperial College, London, developed ideas for a parallel machine to support the functional language, Hope[2]. In 1983 a joint collaboration between Imperial College and ICL started, which would further develop and build three prototypes of ALICE.

ALICE (Applicative Language Idealised Computing Engine) addressed a number of research areas:

- Based on graph reduction principles.
- Extensible hardware architecture.
- Linear speed-up as processors added.
- Support functional languages. Logic language support was added later.
- Real address memory mapped.
- Utilised simple work distribution scheme.
- Dynamic "garbaging" (or freeing) of used packets.

ALICE is a flexible research vehicle which emulates the ideal architecture. The machine principles are described later.

Stage 2

The next stage was to address further issues and to look for efficiency optimisations to produce a competitive machine. Imperial College have within Flagship continued with an emphasis on language work and compilers. ALICE continues to be developed and is used to investigate program transformations and compiler optimisations for a parallel machine, amongst other work.

Flagship machine development continues mainly between Manchester University, led by Ian Watson, and ICL. Manchester University have experience of developing a number of processors including the Manchester Dataflow Machine. This is a high-performance parallel machine, but lacks some of the attributes required for efficient, general-purpose declarative computation.

The first step has been to build a number of simulators known as the Implementation Reference Model (IRM); these are written in the "C" language. Areas being addressed are:

- Graph reduction supporting DACTL[3].
- Sophisticated mechanism for distribution of work between processors.
- Virtual storage based on global store addressing, i.e. all processors can address any part of the distributed computational store.
- Efficient copy mechanism of data from one processor to another.
- Closely coupled processor and store pairs.
- Emphasis on efficiency and performance.
- Distributed I/O sub-system.
- Support for priority mechanisms.
- Variable length packets.
- Caching to reduce store access time.
- Support for a multi-user environment.
- Resilience to individual node failure.

In parallel with the above research an emulator, based on the Motorola 68020 micro-computer, is being developed. This initially has 10 processors with the ability to add many VME bus cards. The IRM simulators will be mapped onto the emulator and research continued in the above areas. A software environment with selected applications will be run on the emulator.

The Flagship machine is described later in this paper.

Stage 3

A third stage has now started. This is a VLSI design study investigating the hardware implementation of the Implementation Reference Model. Initially, this will feedback into the IRM research and prove that the solutions to the research issues can be implemented.

Stage 4

The final stage will be to finalise the VLSI design and simulate, validate and build it.

3 Flagship forerunner – the Imperial College ALICE machine

3.1 The ALICE machine – general

The ALICE processor (Fig. 1) is hosted by an ICL 39/30 mainframe which provides application load facilities and is used also for bulk application output and dump analysis. A Stride workstation provides the diagnostic interface required to start and stop the ALICE machine. Diagnostic facilities for interrogating the machine state and for the interactive tracing of graphs are also provided. The workstation provides the route for loading the code

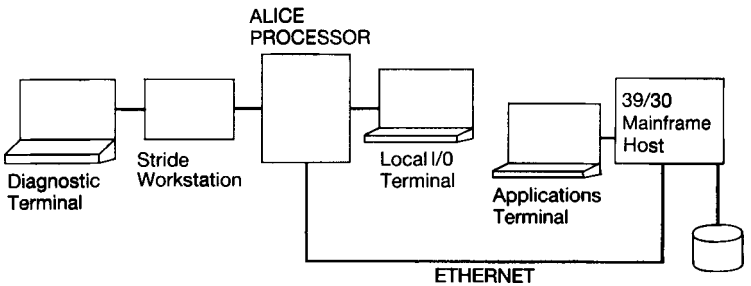


Fig. 1 The ALICE system

into the ALICE processor that programs ALICE to interpret the ALICE architectural model.

ALICE is currently a single user system. It is first loaded with its operational code from the Stride workstation. An application is prepared in the 39/30 host and then loaded via Ethernet into the ALICE processor. The application runs and the user can interact with it via the local I/O terminal. Any bulk output, including dump data, is passed via Ethernet to the 39/30 host, where detailed analysis can take place. ALICE is a research prototype and users are typically interested in how their applications run and perform on a highly parallel machine.

3.2 The ALICE parallel processor

The principles behind a packet-based graph-reduction parallel processor are described in [1]. A very brief description is given below.

3.2.1 Packets: A packet holds information of this nature:

Identifier	Function	Argument List	Secondary Fields

Identifier: the packet's unique address. For ALICE this is a real address.
Function: function associated with this packet. Functions can be primitive such as +, −, log or more complex user-defined functions such as Quicksort or Payroll.
Argument list: pointers "down the graph" to other packets or values.
Secondary fields: control information for evaluation and backward pointers "up the graph".

3.2.2 Graphs: A functional language consists of a number of functional expressions and data structures. These can be represented in graphical form. A graph can be thought of as having a similar topology to a tree data structure, but it differs in that it may have data shared within the graph: for example, Fig. 2 shows packet D shared by packets B and A.

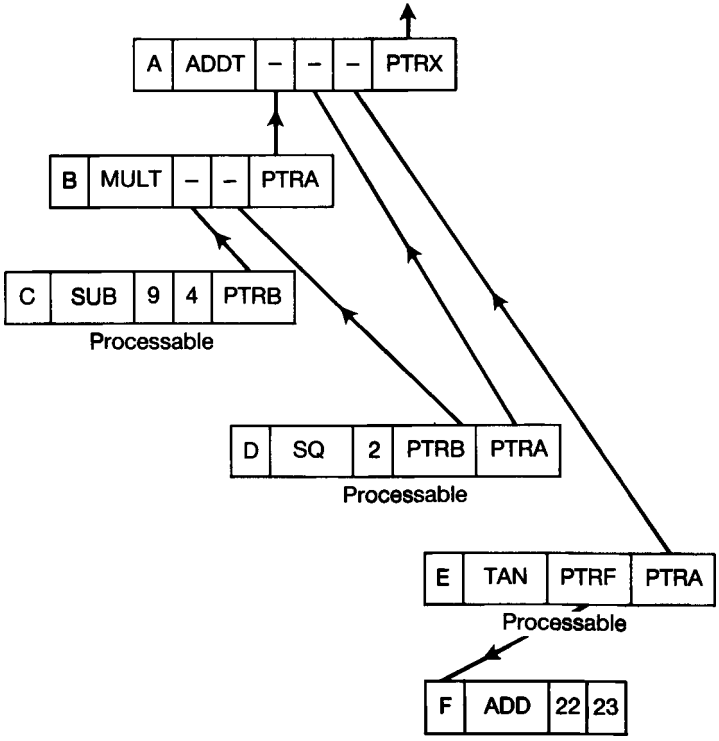


Fig. 2 Graph for packet evaluation of $(9-4)*2^2+2^2+\tan(22+23)$ = ADDT{[MULT(SUB 9,4), SQ 2], SQ 2, TAN(ADD 22,23)}

An application is compiled into graphical form. Functional expressions within the graph are transformed by over-writing the sub-graph with the result of the function application. This may result in the graph increasing or reducing in size – it is best explained using a simple example.

Graph example. This is a simple (and not very realistic) mathematical expression which reduces to a single integer without the graph increasing in size. More realistic examples would typically manipulate complex data structures and would probably include the use of recursive calls of functions within the graph.

The expression $[(9-4) * 2^2] + 2^2 + \tan(22+23)$ can be written

Addt{[mult(sub 9,4), sq 2], sq 2, tan(add 22, 23)}

where the function Addt sums three arguments. This can be represented graphically as in Fig. 2, where the boxes represent packets and, as defined above, the first field is the packet identifier, the second is the function and subsequent fields are either values or forward or backward pointers to other

packets. Note that in this example the result of evaluating packet D is shared by packets A and B. The argument of the tangent function is assumed to be degrees.

Such an expression could be compiled into a number of forms with all the pointers forwards or backwards and a variety of evaluation strategies could be developed.

For our form of the graph an intelligent compiler might make packets C, D and E available for processing and the remaining packets suspended. Three ALICE processors could process these packets in parallel.

The graph would now be:

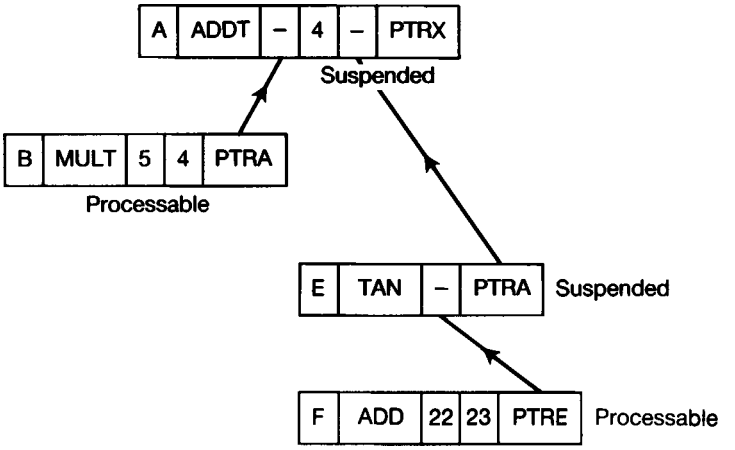


Fig. 3

In Fig. 3 the C and D packets have applied their functions and returned the resulting values to packets B and A. Packet B now has all its values and it can be made available for processing. Packet A does not have all its values and will remain suspended. Packet E, which was processed, does not have its value and it now demands packet F, which will now be processable. Packet E is made suspended. Packets B and F can be processed in parallel. The graph now becomes:

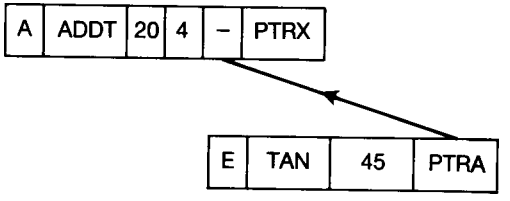


Fig. 4

In Fig. 4 packet B has applied its function and returned it to packet A. Packet F has applied its function and returned it to packet E. Packet E would now be made processable as it has its argument and would return its value to packet A as in Fig. 5.

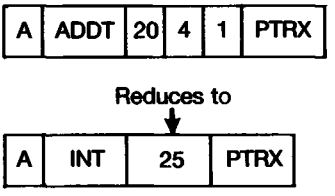


Fig. 5

The answer 25 would be returned to packet X.

Completed packets which no longer hold useful information are "garbaged". Thus packets B to F would all now be automatically garbaged.

3.3 ALICE abstract model

A simple abstract model to perform the above task could be represented as in Fig. 6:

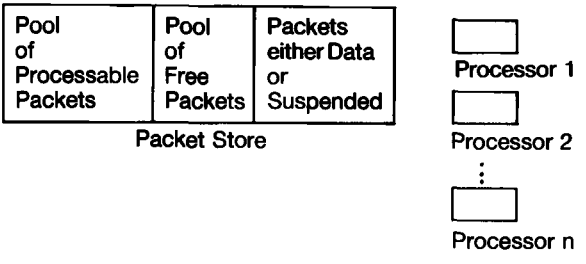


Fig. 6 ALICE abstract model

Processors would all have direct access to any area of the packet store. They could fetch packets from the pool of processable packets, reduce them and write new packets or modify old ones in the packet store. An intelligent packet store could detect when packets had all the required arguments and make them processable. It could also detect when packets were no longer required and mark them for garbaging.

There is no reason why the packet store should be a single entity. It can be partitioned and distributed within the machine.

A communication medium between the processors and the packet store is required: this is the "delta network" shown in Fig. 7.

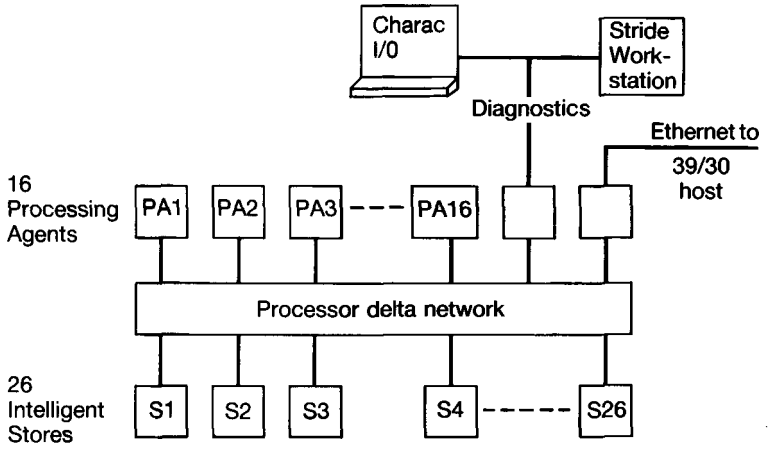


Fig. 7 The ALICE parallel processor

3.4 The ALICE prototype

Figure 7 shows the general topology of the ALICE processor. There is a high bandwidth communication network called a delta network which can connect any processor to any store. Also any store can be connected to any processor. Stores and processors can communicate via the network to the diagnostic system or via the Ethernet to the 39/30 host. There are 16 processing elements and 26 distributed stores.

ALICE uses the Inmos Transputer as its major building block. This is a high-performance 32-bit micro-computer with 2K bytes of internal memory and 2^{32} available addresses for external memory. It has four 1 Mbyte/sec serial links that can be used to enable direct communication between Transputers.

3.4.1 The processing agent: This is shown in Fig. 8; it uses five Transputers, TP 1–5 in the figure.

The units ERU, ETU provide interfaces to the delta network and help decouple the processing agent from the network, to reduce hold-ups.

The units PRU each have 64K bytes of local memory and are used to rewrite packets which are fetched from the distributed store. In conventional machines the majority of functions that the processor provides are resident in fixed micro-code. By contrast, in ALICE the languages that it supports generate many unique user-defined functions and these are fetched and cached by the Function Definition Unit (FDU) as they are required.

3.4.2 The intelligent store: This is shown in Fig. 9; each store uses 2 Transputers and has 2 Mbytes of dynamic memory.

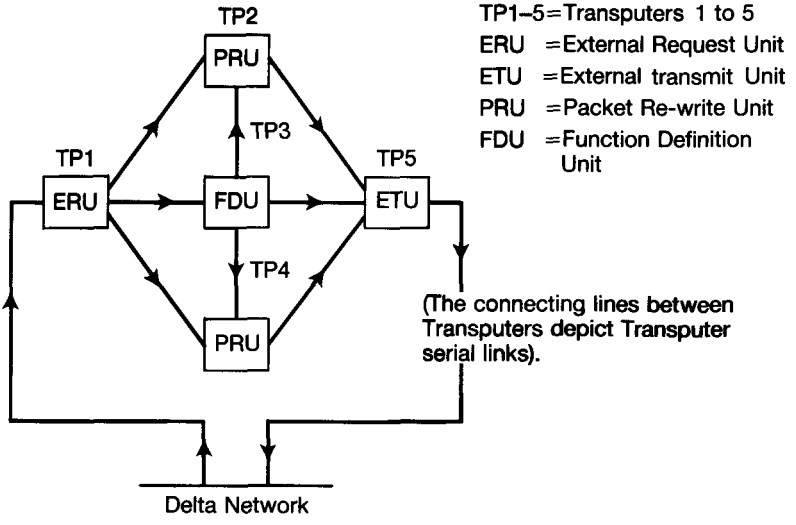


Fig. 8 The ALICE processing agent

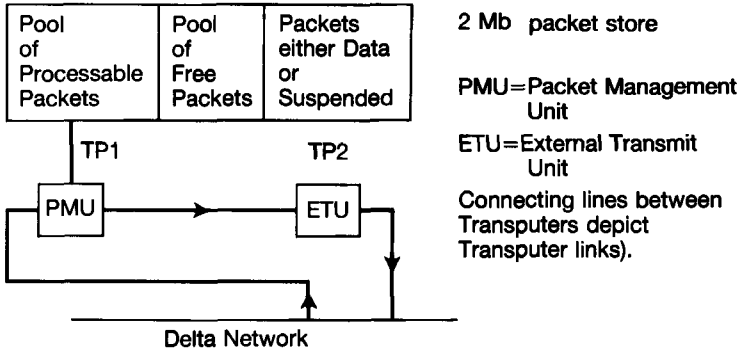


Fig. 9 The ALICE intelligent store

The External Transmit Unit (ETU) decouples the store from the delta network, to reduce hold-ups. The Packet Management Unit (PMU) receives messages from the processors, either to modify packets within main memory or to provide packets to the processors. It can check packet field values and make local decisions about how processing should proceed.

3.4.3 The delta network: Ideally the communication network would permit any processor to communicate with any store, and vice versa, without any clashes or contentions. This would require a full crossbar unit, but unfortunately the interconnect requirement makes this almost impossible to implement. The means provided, the delta net, can be viewed simply as a crossbar that allows clashes to occur. It is made up of 4×4 switches, as shown in

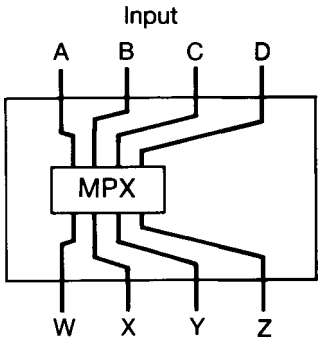


Fig. 10 4 x 4 switch

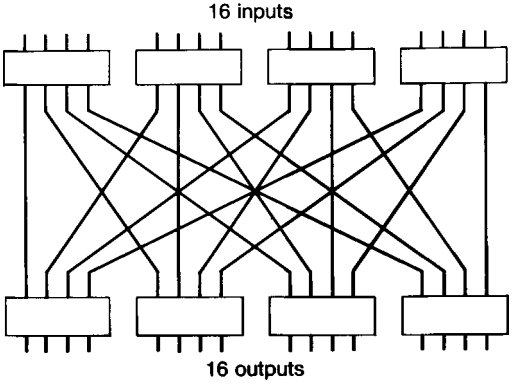


Fig. 11 16-way network

Figs. 10 and 11; a 4 x 4 switch permits any input to connect to any output providing that the output is not already being used for communication.

Figure 11 shows eight 4 x 4 switches used to build a delta network with 16 inputs and 16 outputs. ALICE has a 64 input and 64 output network using the Imperial College-designed 4 x 4 ECL network chip called the XS1. It can be deduced from Fig. 11 that generally if all 16 inputs were to attempt to communicate with all 16 outputs there would be contention for some of the connecting paths. Such contention causes hold-ups and a subsequent loss in usable network bandwidth and additional network latencies.

3.5 How ALICE works

Figure 7 showed the ALICE parallel processor. An application is compiled into *processable packets* and *suspended packets*. The term suspended packet is an over-simplification and has been used here for packets not processable or free. In practice suspended packets can have a number of states. The user-defined function code required to perform the function application when packets are re-written is also compiled into packet format.

These packets are loaded via Ethernet into the 26 intelligent stores, as determined by the compiler. Each processing agent (PA), using a randomly-generated intelligent store number, sends a message via the delta network to that store for a packet to process.

The store, Fig. 9, takes a packet from its processable packet pool and sends it via the delta network to the soliciting PA (Fig. 8). This PA then performs the packet re-write (or reduction) by over-writing this packet and also modifying packets to which this processable packet pointed. New packets may also be generated by obtaining free packet identifiers from the free packet pool.

In the graph example, Fig. 2, then if a PA had received packet C for processing it would perform the subtraction function SUB and write to packet B's third field with the value 5 (Fig. 3). It would then send a message to the appropriate store to say that packet C was now to be garbaged.

All PAs would be re-writing processable packets in parallel.

If the function code required by a packet is not cached in the Function Definition Unit (FDU) in the Processing Agent (Fig. 8), then the FDU fetches the code from the packet store.

An obvious flaw in the above architecture is the latency between processor agents and stores as packets are fetched and modified. This would result in processor agents waiting for stores to reply. To overcome this problem ALICE has a concept of *virtual agents.*

Each physical agent has 16 virtual agents. As a virtual agent awaits packet data from store the PRU process switches to one of the remaining virtual agents which can then also re-write a processable packet. When the first store returns its data the relevant virtual agent is put on a process queue in a PRU (Fig. 8). Research to date shows that 16 virtual agents are adequate to fully-utilise the PRU Transputer, providing that the application has sufficient parallelism.

ALICE has many mechanisms for controlling how graphs are evaluated and for solving synchronisation problems, and has flow control to avoid deadlocks. It has support for both function and logic declarative languages.

3.6 The ALICE language route

The language route is shown in Fig. 12. All languages compile into the ALICE Compiler Target Language (CTL), which describes how to reduce packets for any given function. The Implementation Specific Language (ISL) is CTL-compiled into a byte string for interpretation by the PRUs; these are programmed in Occam, the language designed for the Transputer.

Declarative languages:

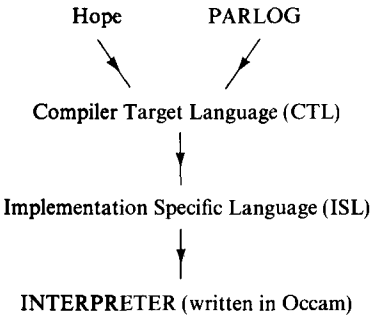


Fig. 12 ALICE language route

3.7 ALICE results to date

The first ALICE prototype was delivered to Imperial College from ICL, West Gorton, in June 1986, after extensive validation in the laboratory; research is continuing in the College.

Highly parallel applications have been found to give a linear speed-up as more processors are added, to within a few per cent of the theoretical maximum; so the principles of a packet-based graph reduction machine have been proven. As an illustration, Fig. 13 below shows the performance increase as processors are added in the case of a highly parallel recursive function called NFIB; the increase is very nearly linear.

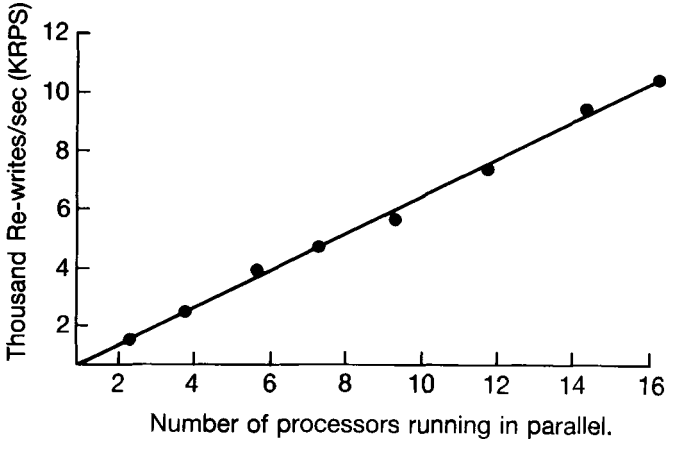


Fig. 13 ALICE performance graph

4 The Flagship machine

Whilst ALICE has met its objectives and proved the feasibility of graph-reduction and that large performance increases can be achieved as further processing agents are added, many further issues need to be addressed.

Issues not addressed by AL ICE

1 ALICE is real-address mapped, therefore applications cannot dynamically use a computational store larger than the RAM mainstore.
2 ALICE distributes and accesses its processable packets using simple algorithms – either random distribution or cyclic. More sophisticated distribution mechanisms are required.
3 It would be better if communication latency was overcome naturally without process switching between virtual agents.
4 ALICE packet store is remote from the processors. Processors with large closely-coupled mainstores would reap the performance advantages that von Neumann machines have. The available packet address range may be viewed as a global address range directly accessible by all processors, but for efficiency local copies could be taken of packets resident in remote stores such that subsequent accesses would be local. This implies that an efficient copying mechanism is needed.
5 ALICE is a single user system. A multi-user system requires protection, priority and process state mechanisms.
6 For efficiency reasons more compact data representations are required than just a small number of arguments of fixed size per packet. Another improvement would be a range of different packet sizes.
7 A distributed I/O sub-system is required to reduce I/O bottlenecks.
8 A Flagship system uses an object-mapped persistent store model. Hardware support is necessary for this.

The abstract model in Fig. 6 is still a valid one with computational store containing processable packets, suspended packets and free packets, and this global store can be addressed by any processor. For efficiency reasons, as mentioned above, it is preferred that processing agents tend to process packets which are in their local stores and only distribute work to other processors when absolutely necessary. When work is distributed, then, ideally, it will be to where associated graph structures live. The example of graph-reduction in Fig. 2 is also still valid for the Flagship machine though evaluation strategies may differ from those utilised in ALICE. Flagship machine topology is shown in Fig. 14.

4.1 Flagship topology

As can be seen in Fig. 14 the Flagship machine will have local I/O controllers providing local disc store, etc. and also connections to a host mainframe.

Conceptually, we now have two networks. The I/O network connects any

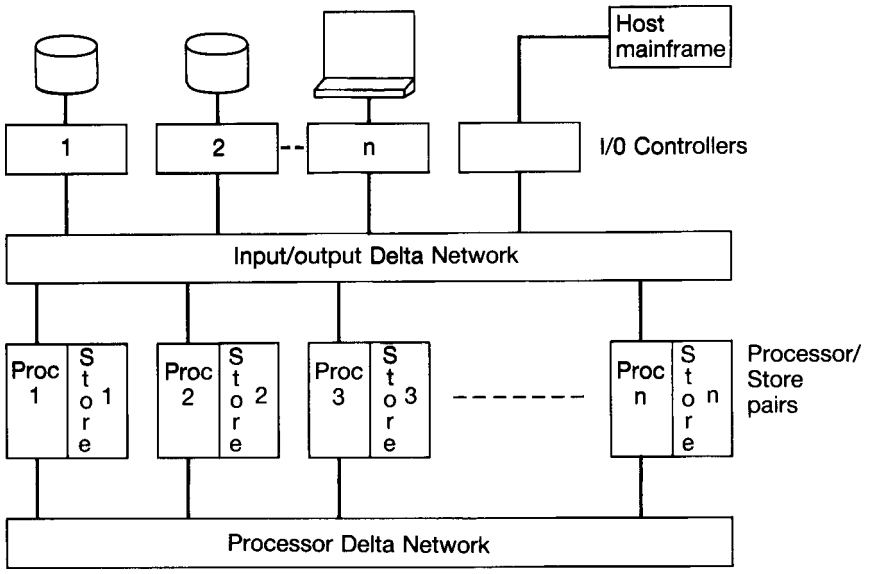


Fig. 14 Flagship machine topology

I/O controller or the host to any processing agent, with many connections active concurrently; and as with ALICE there is a processor network that enables any processor to communicate with any other processor-store unit. But now each processor has an additional direct connection to a local store.

In ALICE the network made a direct connection between a processor and a store before sending the complete message transaction. This is known as circuit-switching. If instead fragments of messages are sent and buffered at each switching stage within the network, contention is reduced because network paths are freed sooner than in the circuit switch: this is called packet switching. Flagship is likely to use hybrid circuit/packet switching in its delta network.

4.2 Work distribution

For processors to distribute work to one another in a sensible way within the Flagship machine it is necessary for the "activity level" of each processor to be known, where this term means a measure of the number of processable packets locally available in the processable packet pool of that processor/store pair. Processors with many processable packets can then distribute some of these to those processors with few. As the processor delta network connects all the processors together it seems sensible that it should be used to distribute the activity levels so that algorithms for sensible distribution of the work can be implemented.

Figure 15 shows the arrangement. Each processor sends its current activity

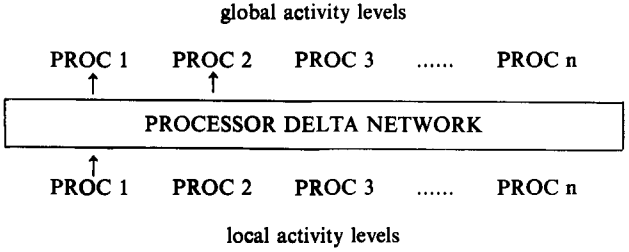


Fig. 15 Work distribution

level into the network and receives an average of all the activity levels; it can then detect if it has more or fewer processable packets than the average, and can distribute work accordingly. Each switching node within the delta network computes its local global average and propagates this backwards; it also remembers which of the paths have the fewest processable packets, so that work can be distributed in that direction. There are variations to the scheme which propagate the lowest rather than the average activity level through the switch.

One objective of the Flagship design is to distribute processable packets, so far as possible, to the processor/store pairs where the data structures associated with each packet reside. This means that processable packets may be sent to processors irrespective of their activity level. This could be unsatisfactory, so a "strength of feeling" is associated with a processable packet when it is distributed and used to modify the effect on packet routing of the global averaging mechanism.

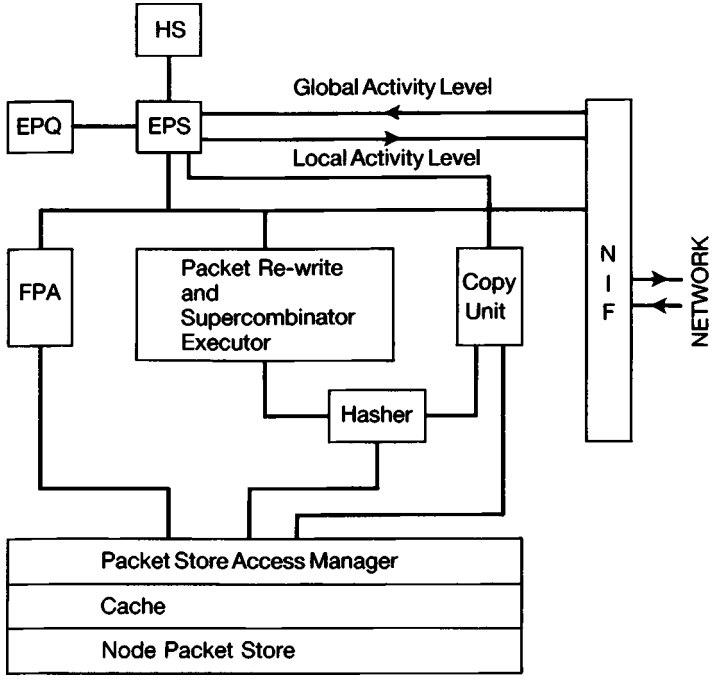
A further complication is one of priority. In a multi-user system processable packets will be of different priorities, which means that high priority packets should be processed first. The activity level mechanism does not include any recognition of this, and therefore the priority levels at which processors are currently running must also be propagated backwards.

4.3 Flagship processing agent/store pair

This is shown diagrammatically in Fig. 16.

4.3.1 Management of the processable packet pool: This is taken care of by the Executable Packet Queue (EPQ), the Executable Packet Scheduler (EPS) and the Holding Stack (HS). We have already seen how the identifiers of processable packets are added to the pool, either as a result of loading compiled packets or dynamically during run time.

If too many processable packets are generated, this can use too much computation store as the graph grows. The EPQ queues a limited number of processable packets on a first in first out basis, i.e. a queue; which means that the graph is explored breadthwise, generating much potential parallelism.



HS =Holding Stack
EPS=Executable Packet Scheduler
EPQ=Executable Packet Queue
FPA=Free Packet Addresses
NIF=Network Interface

Fig. 16 Flagship processing agent/store pair

When the EPQ is full, processable packets that are generated subsequently are placed on the Holding Stack (HS), which provides for Last In First Out processing. This tends to cause the graph to be explored depth first, with less parallelism generated. The EPS schedules the mechanism. New processable packet identifiers are generally passed to the EPQ and HS from the Packet Re-write and Super-combinator unit shown in the centre of Fig. 16; when this unit requires the identifier of a processable packet, for processing, it requests an identifier from the EPS.

The EPS has direct connections to the global network interface in the form of the global and local activity levels shown in Fig. 15. This enables it to cause processable packets to be sent to remote processors to give efficient work distribution.

4.3.2 Packet Re-write and Super-combinator Executor: This is the unit that performs re-writes (or reductions) on processable packets. By following the rules of the Flagship computational model it can fetch, inspect, modify

packets as well as generate new ones. The rules recognise the state of a packet, i.e. if it is processable, suspended (or other states not described in the brief descriptions given), how many arguments are still required and control fields which control the execution strategy. These rules are independent of the function specific to the packet.

A Super-combinator Executor inspects the packet's function field and performs the function application with the relevant value fields from the packets. It then re-directs pointer fields and modifies value fields as was seen in the graph example in Fig. 2. It can also generate new packets. The identifiers to be used for new packets are received from the FPA, described next.

4.3.3 Free Packet Address (FPA): This unit maps the identifiers of the free packets in packet store. It provides new identifiers for the Packet Re-write and Super-combinator Executor unit or collects the identifiers of "old" packets which have been garbaged.

4.3.4 Copy and hasher units: As mentioned earlier a copy mechanism is required for when packets which need to be accessed during a packet re-write are not in the processor's local store.

The copy mechanism causes copies of the packets in a remote store to be made into local store. The copied packet's identifier is stored in the hasher at an address which is a hash of the copied packet's identifier. When further accesses are made for the copied packet, the hasher signifies a hit in local mainstore. This hashing mechanism may also be useful for virtual address translation, rapid access to system tables and caching mechanisms.

To avoid having the processor idle while packets are copied from remote stores the graph is modified to await the remote data and the processor fetches another processable packet for re-writing.

4.3.5 Network Interface (NIF): This provides the network interface to the delta network. Messages with packet information are sent and received to or from other processors. The unit contains message buffers and drives the network protocols, including the work distribution signals previously described.

Only one network interface is shown. Though in concept there are independent I/O and interprocessor networks, currently one physical network is assigned for both tasks.

4.3.6 Packet Store Access Manager, Cache, Node Packet Store: The Node Packet Store is made up of dynamic RAM and constitutes the main packet store. As accesses to the dynamic RAM will take a number of processor beats, a cache made up of static RAM will be included for rapid access of packets in one processor beat. The packet store access manager is

the store interface to the other units; it will provide address translation from virtual to real address and some of the user data protection checks required by multi-user systems.

4.4 The Flagship emulator

An emulator has been built with the topology described in Fig. 14. It uses Motorola's 68020 micro-processors with, initially, 10 megabytes of local store per processor. There are 10 processor store pairs and the diagnostics are provided by a Sun workstation. It will have an ICL 39/30 mainframe host attached later in its development. Initially it will use the propriety VME bus for interprocessor communication. A custom-built delta network is being developed and will replace the VME bus. Only half of the cabinet slots are currently used; the remaining slots will be used to provide the distributed I/O sub-system.

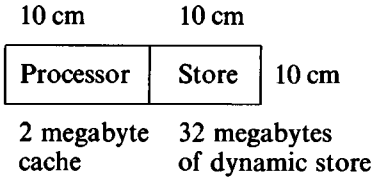
By providing extensive instrumentation and tracing in the emulator we will be able to validate the Implementation Reference Model and machine principles with real applications and the Flagship software environment. With 10 processors, running in parallel genuine concurrency will be obtained at a performance above the levels which simulations on a conventional mainframe could provide.

4.5 Physical implementation of the target Flagship machine

The technology currently considered for Flagship is VLSI at 1·5 micron and 400 000 transistors. It will use a high density interconnect technology. The Packet Re-write and Super-combinator unit requires the attributes of a RISC processor and a propriety 32-bit RISC processor may be suitable. A further two custom VLSI chips will be required.

High density interconnect technology will mount approximately 80 chips on a 10 cm by 10 cm substrate. The 3 VLSI chips plus 2 megabyte cache and associated interface chips will fit on one such substrate.

Mounting 4 megabit dynamic RAM chips on both sides of another 10 cm by 10 cm substrate will provide 32 megabytes of main storage, as shown below.



One hundred and twenty-eight such processors would fit in a cabinet. Two such cabinets could be closely coupled together to utilise 256 input by 256 output delta nets.

The size of the I/O sub-system would depend upon the application domains at which such a processor was aimed.

A range of systems between 8 and 256 processors would be possible.

Typically, these machines would be hosted by an advanced workstation for smaller systems and by a mainframe for the larger systems. Stand-alone systems could be built, given a complete software environment with adequate user interfaces and tools.

4.6 Flagship performance

Quoting performance figures for parallel machines is difficult because as yet there are few, if any, recognised benchmarks for such machines. Benchmarking is itself an area of research.

Functional language performance is often measured in Reductions Per Second (RPS) and logic performance in Logic Inferences Per Second (LIPS).

Each Flagship processing agent will perform at greater than 150 000 RPS. A parallel processor with 256 processing agents, given applications with sufficient parallelism, should have a system power of greater than 40 million RPS.

What is this in terms of Millions of Instructions Per Second (MIPS)? MIPS used for measuring conventional machines can be misleading. Microprocessor designers often claim 10 MIPS for their machines whilst a much more powerful mini-computer mainframe is quoted at only 2 MIPS.

Each Flagship processing agent will be as powerful as a top-of-the-range mini-computer. The total performance for a 256 processing agent Flagship parallel computer should be greater than 500 "mainframe" MIPS.

5 Conclusion

Despite progress to date there is still much research and development work required before viable packet-based graph reduction machines can be made commercially available.

It has been shown that new mechanisms are required for both the distributing and copying of work and data. Also high bandwidth, intimate communication networks are required between processors. Functionality concerned with parallel processing that is additional to the equivalent conventional machine needs to be over-lapped with the basic processor's functionality. This would mean that a single processor would compare favourably with a state-of-the-art von Neumann machine.

From a processor designer's point of view many current techniques used by

mainframe and micro-computer designers are still relevant. Pipelining, caching, pre-fetch of code and data, interrupt and protection mechanism, stacks, etc. are all techniques that are likely to be useful in this style of parallel machine.

The packet-based graph-reduction architecture also lends itself towards new optimisations. A packet can have many argument fields and hold much type and state information. A rule-driven system following the Flagship computational model would enable many packet fields to be generated or modified in parallel in a single processor and in one processor clock beat. Thus new packets could be generated in a single clock beat, including the application of primitive functions (+, −, etc.). This would be equivalent to many von Neumann instructions in a conventional machine.

Research aided by simulation and use of the emulator continues. We hope to generate solutions to the outstanding issues and, by instrumenting our designs running relevant applications, find where to optimise the machine design for cost-effective use of the available silicon.

Unlike conventional machine designers we have a choice. For a given performance we can have lots of processors of relatively simple design, or go for more sophisticated designs with fewer processors. Overall, cost/performance and development risk will determine which option we take.

References

1 WATSON, I., SARGEANT, J., WATSON, P. and WOODS, V.: Flagship computational models and machine architecture. ICL Technical Journal **5**(3), 555–574, May 1987.

2 DARLINGTON, J.: Software development using functional programming languages. ICL Technical Journal **5**(3), 492–508, May 1987.

3 GLAUERT, J.R.W., KENNAWAY, J.R. and SLEEP, M.R.: DACTL: A computational model and compiler target language based on graph reduction. ICL Technical Journal **5**(3), 509–537, May 1987.

GRIP: A parallel graph reduction machine

Simon L. Peyton-Jones, Chris Clack and Jon Salkild
University College, London

Abstract

GRIP – Graph Reduction In Parallel – is a Fifth Generation machine designed to execute functional languages in parallel, using graph reduction. Its design and construction is a project funded by the Alvey Directorate as a collaborative project between University College London, ICL and High Level Hardware Ltd. It is expected that a working prototype will be completed during 1987. The paper gives a brief outline of the principles of graph reduction and of GRIP's architecture.

1 Functional languages and graph reduction

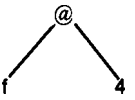
Functional languages are Fifth Generation programming languages which address two of the major current challenges to computer science, those of correctness and parallelism.

The challenge of correctness is the difficulty we experience in writing large, correct programs, a problem which Hoare eloquently outlines in his Turing award lecture[4]. Darlington gives an introduction to functional programming languages[2], showing how their use can alleviate some of these problems.

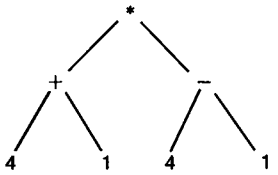
The organisation of a number of independent processors to co-operate in the execution of a single program is the challenge of parallelism. Functional languages contain inherent parallelism and so are a suitable medium in which to express parallel programs. To understand where the parallelism comes from we will look at a functional program:

```
let f x = (x + 1) * (x − 1)
in f 4
```

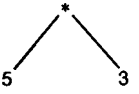
The "let" defines a function "f" of a single argument "x", which computes "(x + 1) * (x − 1)". The program executes by evaluating "f 4", that is, the function "f" applied to 4. We can think of the program like this:



where the "@" stands for the function application. Applying f to 4 gives



We may now execute the addition and the subtraction simultaneously, giving



Finally we can execute the multiplication, to get the result

15

From this simple example we can see that

(i) Executing a functional program consists of evaluating expressions.
(ii) A functional program has a natural representation as a tree, or more generally a graph.
(iii) Evaluation proceeds by means of a sequence of simple steps, called *reductions*. Each reduction performs a local transformation of the graph, hence the term *graph reduction*.
(iv) Reductions may safely take place simultaneously since they cannot interfere with each other.
(v) Evaluation is complete when there are no further reducible expressions.

Graph reduction is described in detail by Peyton-Jones[5].

GRIP – Graph Reduction In Parallel – is designed to execute functional programs by performing parallel graph reduction. Despite the opportunities for parallelism offered by functional languages, only the ALICE project at Imperial College has so far attempted a parallel implementation in custom hardware[1,3]; the main features of ALICE are described briefly in the paper by Townsend in this issue[6]. GRIP is intended to provide state-of-the-art performance at moderate cost by extracting the maximum performance from a fast bus. This means that within its performance range GRIP should provide more power for unit cost than more extensible designs, such as ALICE. Our performance target for a fully populated GRIP is one million reductions per second.

2 The GRIP architecture

Most proposals for parallel graph reduction machines look like Fig. 1. The Processing Elements (PE) traverse the graph held in the Intelligent Memory Units (IMU), discovering reducible expressions and reducing them. The

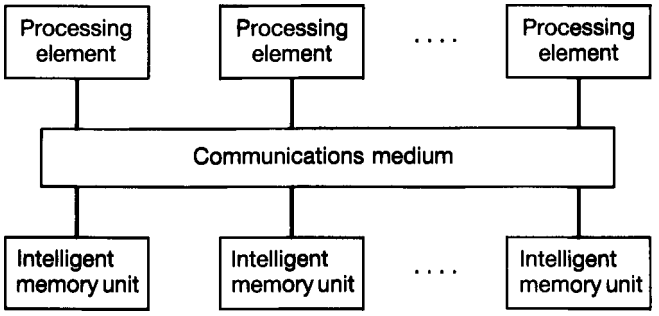


Fig. 1 Physical structure of a parallel graph reduction machine

principle variation between different designs lies firstly in the choice of communications network and secondly in the intelligence in the IMUs.

2.1 The bus

In the case of GRIP we have chosen to use a fast bus, the IEEE Futurebus, for the communications network. A bus offers an extremely cost-effective switch, but at the cost that only one transaction can take place between a PE and an IMU at once, thus limiting concurrency. This places a fundamental limit on the parallelism achievable but gives an extremely cost-effective solution up to this limit. In the case of GRIP we expect to be able to integrate up to 80 or so PEs, on 20 boards, before running out of bus bandwidth and physical space.

The use of a bus allows us to address one research issue, that of parallel reduction, at a time, rather than try to solve several difficult problems at once. By using a bus therefore we expect to exploit a cost/performance/concurrency "window" and to build a working prototype in the relatively short time of 2–3 years.

2.2 The Intelligent Memory Units

GRIP's IMUs will each consist of 5 megabytes of RAM arranged in 40-bit words, with a simple bit-slice microprogrammable processor on the front. Instead of supporting just READ and WRITE commands as normal memories do the IMUs will support an instruction set of high level operations, chosen to support parallel graph reduction. These operations are the unit of indivisibility supported by GRIP (all concurrent machines must provide some indivisible operations to ensure correct synchronisation of parallel activities). In addition, the use of high level operations reduces the requirement for bus bandwidth for communication with the IMUs.

2.3 The Processing Elements

The PEs are autonomous units responsible for performing reductions on the graph held in the IMUs. They will be of straightforward design, based

around a microprocessor, the MC68020, and will include their own private memory which is inaccessible to the rest of the system. The processor within a PE executes a program held in local memory.

2.4 Physical arrangements

Although the PEs and IMUs are logically separate we shall integrate several PEs, and one IMU on each board plugged into the bus. This maximises the use of the bus slots, which are our scarcest resource, and also the number of concurrent activities in the system by putting several on each board.

Most transactions between a PE and an IMU will be carried out on a split cycle basis, to make the best use of scarce bus bandwidth. The PE will write a transaction request into a fast transaction buffer held in the bus interface section. The bus interface will then acquire the bus (which may take some time), send all pending requests to the corresponding buffer on the destination board and relinquish the bus. The request will be processed by the recipient IMU which will write a reply transaction into the transaction buffer, and this reply will then get transferred back to the requesting PE by the same mechanism.

Achieving an implementation of this protocol without imposing substantial latency on transactions is one of the major hardware challenges of the project.

3 Project status

The GRIP project is funded by the Alvey Directorate as a collaborative project between University College London, ICL and High Level Hardware Ltd. Three full-time Research Assistants form the main team based at UCL and led by Simon Peyton-Jones. Work began in the late autumn of 1985; construction of the machine is now under way and the expectation is that a prototype will be working during 1987.

The GRIP architecture is described in more detail in Peyton-Jones[7] and Peyton-Jones et al[8].

References

1 DARLINGTON, J. and REEVE, M.: 'ALICE – a multiprocessor reduction machine for the parallel evaluation of applicative languages'. Proc. ACM conference on functional programming languages and computer architecture, New Hampshire, Oct. 1981, 65–75.
2 DARLINGTON, J.: 'Functional programming'. In: Distributed Computing, Duce, D.A. (editor), Peter Peregrinus, 1984.
3 DARLINGTON, J.: 'Software development using functional programming languages'. ICL Technical Journal Vol.5, No. 3, 1987, 492–508.
4 HOARE, C.A.R.: 'The Emperor's old clothes'. CACM, Vol. 24, No. 2, 1981, 75–83.
5 PEYTON-JONES, S.L.: 'Implementation of functional programming languages'. Prentice-Hall (to be published March 1987).

6 TOWNSEND, P.: 'Flagship hardware and implementation'. ICL Tech. J., Vol. 5, No. 3, 1987, 575–594.
7 PEYTON-JONES, S.L.: 'Using Futurebus in a fifth-generation computer'. Microprocessors and Microsystems Vol. 10 No. 2, March 1986
8 PEYTON-JONES, S.L., CLACK, C.D., SALKILD, J. and HARDIE, M.: 'GRIP–a high-performance architecture for parallel graph reduction', Internal Note 2079, University College London (submitted to IFIP Conference on Functional Programming and Computer Architecture, Portland, 1987).

Notes on the authors

Professor M.P. Atkinson

Malcolm Atkinson is a Professor of Computing Science at the University of Glasgow with previous appointments at the universities of: Pennsylvania, Edinburgh, East Anglia, Cambridge, Rangoon and Lancaster. He began his computing science career building language processors. He moved into computer aided design and specialised in the provision of databases for generic CAD applications. This led to the current work on eliminating discontinuities in the programmer's environment.

E. Babb

Ed Babb obtained qualifications in Electrical Engineering and Computer Science from Imperial College. He then researched adaptive pattern recognition systems at Cambridge University on secondment from Hawker Siddely Dynamics. From about 1971, working in ICL, he researched speech recognition and information retrieval. His work on CAFS covered storage structures, architectures and query languages. He is now studying the application of mathematical logic to business and is currently the manager of the logic language project in the Systems Strategy Centre at Bracknell.

C.W. Bartlett

Clive Bartlett took a degree in Electrical Engineering at King's College, Newcastle and since that time has worked exclusively in the computing industry–starting with EMI and subsequently with ICL. Initially he was concerned with hardware design, both of logic elements and of computer subsystems. In 1965 he moved over to software and worked on Test and Diagnosis problems at all levels, culminating in being involved in the initial work on Testing and Error Management for the 2900 Series and being responsible for the overall design, at system level, of the Test and Diagnostic Systems for several members of the 2900 Range. In 1979, turned to working on tools systems for the numerate management of quality including a model for the prediction of system reliability based on the use of Expert System techniques. In 1982, became a founder member of the (then 4 strong) new Knowledge Engineering Group. Since then has, apart from writing S39XC, worked on a wide range of KBS topics at all levels. He has had an interest in Artificial Intelligence and its application to practical problems since 1968, when a friend sought advice on how to write a program to solve problems

concerned with the design of conveyor belt systems from kits of standard parts–a problem which has only recently become solvable by the use of techniques related to those described in his paper.

J.B. Bocca

Jorge B. Bocca graduated from the Universidad de Chile with a degree in Economics. After graduating he worked as a System Analyst in industry and then, he returned to University at St. Andrews to do an M.Sc. in Computational Science. In 1979 he joined Southampton University to do research in the field of relational data base systems. Here, he obtained his Ph.D. in Computer Science. From there he went to Bristol University to work as a researcher into distributed data base systems and later on, to the University of Ulster as a lecturer in Computer Science. Shortly after joining ICL early in 1985, he was detached to the European Computer-Industry Research Centre (ECRC) in Munich, as a researcher in the field of Knowledge Bases. He has numerous publications in the fields of logic and data bases.

P. Broughton

Phil Broughton has a B.Sc. in Electronics Science from Southampton University. After graduating in 1970 he joined ICL to work on the hardware design of large and medium mainframe machines; as a logic designer he was one of the key engineers for the 2980 and its development into 2982, later working on the design of a large pipelined successor to the 2980. Since then he has managed the design of the 3900 HSC, LSC & MSC and level 30 Store. On Flagship he has managed the early phase of hardware development, later moving to his current post to manage the system software.

Chris Clack

After graduating from Queen's College Cambridge Chris Clack spent two years in the oil industry as an International Field Engineer before moving into computing. He has Master's degrees in both Physics and Computing Science and has worked for the last three years as a member of the research staff at University College London. His major research interests are functional languages and parallel architectures. He is a consultant to The Instruction Set Ltd, specialising in UNIX and C. He co-designed the graph reduction machine GRIP and is currently working on its construction at UCL as part of the Alvey programme.

M.G. Cutcher

Martyn Cutcher graduated from the University of Surrey in 1981 with a Bsc. (Hons.) in Civil Engineering and joined ICL (Distributed Systems) in August 1982 to work on S25 systems software. In 1985 he moved to Applied Systems to work on the Alvey Flagship project, investigating the Fifth Generation applications scenario. In January 1986 he joined the Systems and Architec-

ture group in Applied Systems where he has worked on the ESPRIT 956 COCOS project and has continued work on parallel logic languages.

Professor J. Darlington

John Darlington studied for his degree in Mathematics and Computing at the London School of Economics from 1966–1969. He then went to the Department of Artificial Intelligence at Edinburgh University to study for a Ph.D. degree with Rod Burstall. The Ph.D., which was awarded in 1973, introduced for the first time the idea of systematic program development using formally based transformations. This work was continued as a Research Fellow until 1977 when Dr. Darlington moved to Imperial College to take up a lectureship. There Dr. Darlington was responsible for the initial invention of the ALICE parallel graph reduction machine. He was promoted to Reader in 1982 and to the personal Chair of Programming Methodology in 1985.

Dr. Darlington's work has been centred around the development of Functional Programming languages particularly topics of program transformation and the design of parallel architectures. He heads a research group developing these topics and is Principal Investigator on the Flagship project, an Alvey collaboration with ICL and Plessey, aiming to develop these ideas commercially.

Dr. J.R.W. Glauert

John Glauert is Information Technology Lecturer in the School of Information Systems at the University of East Anglia, Norwich. After graduating from Cambridge University with a degree in Natural Sciences, he studied for an M.Sc. in Computing Science at Manchester University where he was involved in the design of an early Dataflow Language. He studied for his Ph.D. at Cambridge, researching into Relational Database Systems. He returned to Manchester to work on language design and implementation for the Dataflow Project. At UEA he is researching general graph-rewriting models of computation for parallel systems with a grant under the Alvey Programme.

Dr. J.R. Kennaway

After graduating in Mathematics at Edinburgh University, Richard Kennaway studied at the Programming Research Group at Oxford University for an M.Sc., and subsequently a D.Phil. in Mathematics, in the area of formal semantics of parallelism and non-determinism. He then held a research post at Edinburgh University, where he continued to work in this field and completed his D.Phil. thesis. He is currently working in Professor Sleep's research group, on graph-rewriting semantics for parallel computation.

S.R. Leunig

Steve Leunig has a B.Sc. in Mathematics from the University of Western Australia and a M.Sc. in Computing Science from London University. After

graduating in 1974 he joined ICL to work in VME support, later moving to join the VME Comms team where he became a leading designer. On Flagship he is leading the design of the system software.

Professor R. Morrison

Ronald Morrison is a graduate of the Universities of Strathclyde, Glasgow and St. Andrews. After a three year spell as a system programmer at Glasgow he moved to a Lectureship in the Department of Computational Science at St. Andrews in 1972 where he is now one of the two Professors, and the leader of the Programming Research Team. His main interests are in programming language design and implementation. He has designed several languages, including S-Algol and PS-Agol: he is co-author of two books, Recursive Descent Compiling (with A.J.T. Davie) (Ellis Horwood 1981) and Introduction to Programming with S-Algol (with A.J. Cole) (Cambridge University Press 1982) and has in preparation Developing Large Software Systems using Ada (with I. Sommerville) (Addison Wesley) and Data Types and Persistence (with M.P. Atkinson and O.P. Buneman) (Springer). In 1983 he was Visiting Fellow at the Australian National University, Canberra. At St. Andrews he is one of the technical directors of a £2M project aimed at revolutionising programming techniques, funded by the Science and Engineering Reaserch Council and STC.

Jean-Marie Nicolas

Jean-Marie Nicolas received the M.Sc. degree in Computing Science from the University of Grenoble, France, in 1969, the "Doctorat de 3éme Cycle" and the "Doctorat D'Etat" degrees from the University of Toulouse, France, in 1973 and 1979 respectively.

During 1969–1970 he served as a research assistant with the Computer Science Laboratory of the University of Toulouse. From 1971–1983 he was a research engineer with the ONERA-CERT Computer Science Department in Toulouse and a part-time lecturer at Sup'Aero. Shortly after joining the BULL company in 1984 he was detached to the European Computer-Industry Research Centre (ECRC) in Munich, West-Germany where he is presently in charge of the Knowledge/Data Base research group.

Dr. Nicolas has published several books devoted to artificial intelligence and databases, as well as numerous articles in journals and conferences devoted to these subjects. His primary research interests concern both theoretical and practical aspects of databases, logic/deductive databases and knowledge bases.

Simon L. Peyton-Jones

Simon Peyton Jones is a Senior Lecturer in Computer Science at University College London. After graduating from Trinity College Cambridge he

worked on industrial computing for two years before taking up his present post. His main research interest is in functional programming languages and their implementation, and he is currently leading a team in an Alvey project to design and build a high performance graph reduction machine, GRIP. He has written a book about the implementation of functional languages using graph reduction, to be published by Prentice Hall in March 1987.

E.C.P. Portman

Charlie Portman gained his B.Eng. at Liverpool University and joined Ferranti's Computer Department in 1954. He worked as a circuit engineer, a logic designer, a test programmer and led a drum commissioning team before taking responsibility for the Sirius Project. He worked on the Ferranti Orion Computer and took the first of these machines to Sweden in 1963. He later led development of the 1900 series machines at West Gorton Manchester. In 1972 set up and managed the Software Division for ICT with staff at Manchester, Kidsgrove, Stoke and Stevenage. The Division had responsibility for 1900 Executive Programmes, Test Programmes for both 1900 and the early 2900 machines and also was responsible for an early 2900 Supervisor programme.

Later roles in Systems Engineering and Advanced Development led to the present task of leading the Alvey DHSS Large Scale Demonstrator for ICL's Knowledge Engineering unit. This demonstrator is a five year, £3·75M, advanced development project aimed at introducing IKBS techniques into large legislation based organisations. Over 30 staff in the Universities of Lancaster and of Surrey and Imperial College, London and at Logica and ICL are involved.

Among other external activities he is a member of the Science and Engineering Research Council's Computer Science Committee.

John M. Pratt

John Pratt is the Group Leader of the Man-Machine Interaction Group of the European Computer-Industry Research Centre (ECRC), which is jointly owned by ICL, VLL, and SIEMENS, and is located in Munich. He was an English Electric Scholar at Cambridge, took an Engineering degree, and naturally joined English Electric Computers in 1961. He has been with the emerging ICL most of the subsequent time, with short diversions with Honeywell Controls and ITT Europe. His work has concentrated on the design and engineering problems of input-output systems, with particular emphasis on the efficient integration of software and hardware. He has particularly investigated the special problems of real time system design, data communication systems and high performance interactive graphic systems, and encouraged the investment in MMI research in universities. His current research objective is to include the strengths of the user within the system design process, leading to a more powerful, symbiotic, combinations of man and machine. He is a Chartered Engineer and a member of the IEE.

G.D. Pratten

Graham Pratten is a Chief Research Fellow in Software Directorate, STL North West. He is an MA of Cambridge University, having graduated in mathematics with Tripos Part III in 1962. He followed the LEO III–English Electric–ICL–STC path through the evolution of STC. His career has involved him with LEO III software, System 4 software strategy, the EMAS (Edinburgh University) project, VME database systems, CADES design, mainframes software engineering strategy, ICL's University Research Council and the Software Engineering Technology Centre (SETC).

S. Prior

Steve Prior has worked for ICL since 1969. He has been involved with various aspects of 1900 and 2900 software design and development. He is currently designer/implementer with the Flagship system software team. He is interested in formal methods, having spent four months in 1985 on sabbatical to the Formal Methods group at Manchester University, and is investigating the introduction of formal specification techniques into the Flagship project.

B.J. Procter

Brian Procter has a B.Sc. in Physics and Mathematics. He has worked in the computer industry for nearly thirty years, joining EMI where he worked on one of the earliest transistor "second generaton" machines. With ICT and subsequently ICL, he was the lead designer and later design project leader on most of the smaller 1900 series computers, 2903 and 2960; and system design manager on Series 39 Level 30. Next came an involvement in shaping the VLSI design policy carried out with corporate responsibility. His present job is technical management of the Flagship programme and member of the group technical steering team with special responsibility for new architecture.

M.J. Rigg

After graduating from Cambridge University with a degree in Mechanical Sciences Malcolm Rigg joined IBM as an engineer and subsequently moved to IBM Hursley where he worked on product assurance of both hardware and software. After joining ICL Dataskil he spent some years working with the Commission of European Communities as a technical consultant in APL, graphics and statistical tools. More recently he has concentrated on the evaluation of new products and techniques, and the formulation of technical strategy within the Public Services Business Centre of Industry Systems–formerly the Public Administration Business Centre of Applied Systems.

Jon Salkild

Jon Salkild graduated from Jesus College Cambridge with a degree in Genetics and went on to do an M.Sc. in Computer Science at University College London.

He is currently involved in the design and construction of the parallel graph reduction machine GRIP at UCL as part of the Alvey program.

J. Sargeant

John Sargeant is a SERC research fellow in the department of Computer Science at the University of Manchester. He worked on the Manchester Dataflow project, obtaining his Ph.D. in 1985 for work on data structure implementation in dataflow computers. His research interests include parallel computer architecture, operational behaviour of parallel computer systems, and programming language implementation for parallel machines.

C.J. Skelton

Colin Skelton graduated from University College, London, with an Honours Degree in Physics. He designed servosystems and launch control electronics with Hawker Siddeley Dynamics (now British Aerospace) before joining ICL. Since 1971 he has managed the development of the 2960, 2956, 2966 and most recently the Series 39 Level 30 mainframe systems. As Head of the 5G Systems Department in ICL's Mainframe Systems Division he is now Project Manager for the Flagship declarative architecture and parallel processing consortium.

Professor M.R. Sleep

After graduating in Physics at Bristol University, Ronan Sleep lectured at Brunel University where he also completed his Ph.D. thesis. He then moved to the University of East Anglia at Norwich where he has built up a research team in the field of New Generation Languages and Architectures, supported by SERC and Alvey grants. He was seconded to the Alvey Directorate for a period of two years during which he helped establish the Alvey Architecture Programme. He is presently Professor of Computing Science at UEA.

M. Small

Mike Small is a graduate of Brunel University and has worked for ICL since 1967. He is the System Designer responsible for VCMS (VME Capacity Management System) and Reveal, the Knowledge Engineering tool upon which VCMS is based. Previously he was involved in the design of test systems for ICL mainframes including ME29. He is a Chartered Engineer, a member of the BCS and the BIM.

C.M. Thomson

Tom Thomson received a BA in mathematics at Oxford and an M.Sc. in logic at Bristol (1967). After working for English Electric, the University of East Anglia, and CTL on various aspects of software development he joined ICL in 1971 and specialised in communications for 13 years. Since 1985 he

has acted as a system architecture consultant in ICL's Fifth Generation system development team.

Paul Townsend

Paul Townsend joined the Royal Air Force in 1963 for a three year electronics and navigation apprenticeship with the 104th entry RAF Malton. His subsequent main involvement was with digital/analogue flight simulators.

After working with Ferranti he joined ICL Mainframe Systems in 1974, became involved with the development of the 2980 and 2982 large mainframes and with their proposed successor as a project leader and then was a Design Manager on the Input/Output systems for the System 39 Levels 30 and 80 machines. He then became Design Manager for the ALICE project and is now Manager for the ICL Fifth Generation systems computer architecture and hardware.

I. Watson

Ian Watson is a senior lecturer in the Department of Computer Science, University of Manchester. He has been employed at the university since obtaining his Ph.D. there in 1973. His main research interest is in the architecture of parallel computers. He was responsible for the major part of the engineering design of the Manchester Dataflow computer; a novel prototype parallel machine which became operational in 1981. Since then, he has concentrated on the design of machines to support the declarative style of programming.

P. Watson

Paul Watson is a lecturer in the Department of Computer Science at the University of Manchester, where he has been employed since completing his Ph.D. in September 1986. His major research interest is in the design of computational models for the parallel evaluation of declarative languages. Other interests include Functional and Logic Programming, the Lambda Calculus and Garbage Collection. Work in the latter area has led to the design of a new scheme for Garbage Collection on Parallel Computers, which has been patented by ICL.

J.V. Woods

Viv Woods is a senior lecturer in the Department of Computer Science at the University of Manchester where he has been employed since 1968. He had previously been employed for five years by ICT before moving to an academic post at Manchester to undertake research work for a Ph.D. He was heavily involved in the MU5 project undertaken in liaison with ICL and subsequently with the MU6 development. Subsequently his interest has transfered to the architecture of parallel machines and is currently investigating designs to support declarative programming styles.