Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document ; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1987

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
ABSTRACT

These Proceedings contain written versions of lectures delivered at the 1986 CERN School of Computing. Two lecture series treated trends in computer architecture and architectural requirements for high-energy physics. Three other courses concentrated on object-oriented programming, on objects in Ada, and on modularization and re-usability. Expert systems, their applications, and knowledge engineering for graphics were treated by three lecturers. The programme of the School covered, in addition, practical aspects of networks, buses for high-energy physics, the design of data-acquisition systems, and examples of on-line systems for particle physics experiments. Optical storage methods, software for distributed systems, symbolic formula manipulation, and solid modelling and rendering were also covered. Experience with transputers was the subject of an unscheduled presentation.
PREFACE

The 1986 CERN School of Computing was held in Renesse, on the island of Schouwen-Duiveland in the Netherlands, from 30 August to 13 September. It was the ninth in the series of Computing Schools organized by CERN. Fifty-four participants from 14 countries followed the very interesting programme, which consisted of 47 lectures. Participants also had the possibility of hands-on experience in computer graphics, using ten work stations installed at the School.

The School was sponsored by NIKHEF-H, Amsterdam and Nijmegen. The substantial financial contributions of NIKHEF-H, the Stichting Fundamenteel Onderzoek der Materie, the Nederlandse Natuurkundige Vereniging, the Stichting Physica and the Koninklijke Academie van Wetenschappen are gratefully acknowledged.

The Organizers wish to sincerely thank the local organizers for their efforts: Dr. W. Hoogland, Professor L.O. Hertzberger and Dr. J. Schotanus. Owing to their efforts we can look back at a School which ran very smoothly indeed. Walter Hoogland was a most efficient co-director.

We were very fortunate in having several distinguished speakers for the opening sessions: Dr. P. Tindemans of the Direktie voor Wetenschapsbeleid of the Ministry of Science and Education, Professor H.B.G. Casimir and Dr. W. Hoogland. All three are warmly thanked for their contributions.

The mayor of the township, Mr. Delst, who also honoured the School with his presence, reminded the participants that they were staying at the centre of a region which was ravaged in 1953 by enormous floods. It was therefore natural to pay attention to the efforts undertaken to prevent such catastrophies in future. A richly illustrated evening lecture by Dr. Theun Bruins of the Ministry of Water Management was greatly appreciated by the participants. An excursion was organized to the very large flood barrier a few kilometres away. Mr. B.W. de Waal Malefijt of the same Ministry gave very interesting explanations before guiding the participants around the impressive works. Both these speakers are warmly thanked for their very stimulating presentations.

The practical sessions on Computer Graphics would not have been possible without the efforts of Irene Seis, who also gave the tutorial introduction and helped the participants. The work stations were put at the disposal of the School by Apollo Computers B.V. and by the Digital Equipment Corporation (DEC). Both firms are sincerely thanked for their gesture. We also wish to thank Mr. van der Wel of Apollo for their financial contribution enabling a social event, and Mr. Skaali for having provided material help from the Geneva Headquarters of DEC.

The hotel 'De Zeeuwse Stromen' proved to be an ideal site for a School, and we extend our warmest thanks to Mr. J. Brongers, the director, and his staff who spared no efforts to make our stay a very pleasant one.

Mrs. Ingrid Barnett looked after the School Secretariat in her usual efficient way, and Mrs. Edith Lafouge assisted with the editing of these Proceedings, with the help of colleagues from the Composition and Printing Group. We gratefully acknowledge their contribution.
Last but not least, we wish to thank the members of the Advisory Committee for putting together such an interesting programme as well as the speakers for bringing this programme to life and for extending it through personal contacts with the students. All readers of these Proceedings will appreciate the quality of their efforts.

C. Verkerk
Editor

* * *

ADVISORY COMMITTEE

BUTTERWORTH, I. CERN, Geneva, Switzerland
CHURCHHOUSE, R.F. University College, Cardiff, U.K.
DELOBEL, C. Laboratoire IMAG, Grenoble, France
HERTZBERGER, L.O. NIKHEF-H, Amsterdam, Netherlands
HOOGLAND, W. NIKHEF-H, Amsterdam, Netherlands
LEVRAT, B. Geneva University, Switzerland (Chairman)
SCHOTANUS, D.J. Nijmegen University, Netherlands
SUMNER, F.H. Manchester University, U.K.
TSICHRITZIS, D. Geneva University, Switzerland
VERKERK, C. CERN, Geneva, Switzerland (Scientific Secretary)
VON RÜDEN, W. CERN, Geneva, Switzerland
WIEGANDT, D. CERN, Geneva, Switzerland
ZANELLA, P. CERN, Geneva, Switzerland
BARNETT, I. CERN, Geneva, Switzerland (School Secretary)
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>v</td>
</tr>
<tr>
<td>CHARACTERISTICS AND PRACTICAL USE OF NETWORKS FOR HIGH ENERGY PHYSICS</td>
<td>1</td>
</tr>
<tr>
<td>B.E. Carpenter</td>
<td></td>
</tr>
<tr>
<td>COMPUTER ARCHITECTURES FOR HIGH ENERGY PHYSICS</td>
<td>14</td>
</tr>
<tr>
<td>R.P. Mount</td>
<td></td>
</tr>
<tr>
<td>WHAT IS THE 'OBJECT' IN OBJECT-ORIENTED PROGRAMMING?</td>
<td>43</td>
</tr>
<tr>
<td>O.M. Nierstrasz</td>
<td></td>
</tr>
<tr>
<td>OBJECTS IN ADA AND ITS ENVIRONMENT</td>
<td>54</td>
</tr>
<tr>
<td>I.C. Pyle</td>
<td></td>
</tr>
<tr>
<td>MODULARIZATION AND REUSABILITY IN CURRENT PROGRAMMING LANGUAGES</td>
<td>65</td>
</tr>
<tr>
<td>P. Klint</td>
<td></td>
</tr>
<tr>
<td>AN INTRODUCTION TO SYMBOLIC MANIPULATION</td>
<td>78</td>
</tr>
<tr>
<td>J. Vermaseren</td>
<td></td>
</tr>
<tr>
<td>BUSES FOR HIGH ENERGY PHYSICS</td>
<td>100</td>
</tr>
<tr>
<td>W. von Rüden</td>
<td></td>
</tr>
<tr>
<td>THE UA1 VME DATA ACQUISITION SYSTEM</td>
<td>124</td>
</tr>
<tr>
<td>S. Cittolin</td>
<td></td>
</tr>
<tr>
<td>MODULAR ARCHITECTURES FOR ON-LINE SYSTEMS</td>
<td>147</td>
</tr>
<tr>
<td>A. Vascotto</td>
<td></td>
</tr>
<tr>
<td>SOFTWARE FOR DISTRIBUTED SYSTEMS</td>
<td>154</td>
</tr>
<tr>
<td>J.C. Vermeulen</td>
<td></td>
</tr>
<tr>
<td>OPTICAL STORAGE</td>
<td>174</td>
</tr>
<tr>
<td>G.J. Verhaart</td>
<td></td>
</tr>
<tr>
<td>INTERFACES AND DATA FORMATS FOR TRANSFER AND COMMUNICATION IN COMPUTE</td>
<td>186</td>
</tr>
<tr>
<td>J.L. Encarnacao and J. Schönhut</td>
<td></td>
</tr>
</tbody>
</table>
INCORPORATING KNOWLEDGE ENGINEERING AND COMPUTER GRAPHICS FOR EFFICIENT AND USER-FRIENDLY INTERACTIVE GRAPHICS APPLICATIONS  
  J.L. Encarnacao

MODELS AND METHODS FOR THE IMPLEMENTATION OF DECISION SUPPORT SYSTEMS AIMING AT THE EVALUATION AND SELECTION OF CAD-SYSTEMS  
  J.L. Encarnacao, L.A. Messina and Z.I. Markov

A PICTORIAL OVERVIEW OF SOLIDS MODELLING AND RENDERING  
  A. van Dam

FROM SHALLOW TO DEEP KNOWLEDGE  
  H. de Swaan Arons

TRENDS IN ARCHITECTURES  
  L.O. Hertzberger

UNDERLYING PRINCIPLES OF EXPERT SYSTEMS  
  C.R. Pellegrini

DATA ACQUISITION SYSTEM DESIGN USING THE STRUCTURED ANALYSIS/STRUCTURED DESIGN (SASD) METHODOLOGY  
  J. Harvey

List of Participants

ANNEX: PARALLEL PROCESSING USING TRANSPUTERS OR OCCAM (OR BOTH)  
  C.R. Askew

Editor's note:  
For technical reasons, the order of the papers in these Proceedings differs from the sequence in which the lectures were delivered at the School. An unscheduled lecture by a participant is included as an annex.

x
CHARACTERISTICS AND PRACTICAL USE OF NETWORKS
FOR HIGH ENERGY PHYSICS

B.E. Carpenter

CERN, Geneva, Switzerland

1. WHAT ARE NETWORKS AND WHAT ARE THEY GOOD FOR?

A network is a very general concept: it is essentially any set of objects joined together by paths along which something can be moved. Thus the road system of the Roman Empire was a network linking cities and army camps, along which soldiers and supplies could be moved. Cities and camps off the roads could not survive because they could not be supplied or defended against barbarians. Thus the benefits of being on the network far outweighed the costs and disadvantages.

The twentieth century depends even more heavily on networks of many types: roads, railways, airlines, shipping, the postal system, telephone, telegram, telex, electricity, oil, natural gas, radio, television, to name only thirteen. These familiar examples can be used to illustrate a few general characteristics of networks:

- The service provided by a network depends heavily on the type of link employed. Roads provide a door-to-door service on demand, but the user has to make an effort. Railways provide city-to-city service at specified times, but the user can relax en route.
- Successful use of a network involves understanding the rules and obeying them precisely. A water hose will not fit a gas tap, and one must not drive on the right in England.
- A distinction may be made between the network itself (the roads) and the objects connected by the network (the houses).
- An object may participate in several networks simultaneously (a house is on at least eight of the above networks).
- Networks may be built on top of other networks. The postal system uses roads, railways, airlines and ships. Special signals may be sent on the electricity network to switch on and off the street lights, a vital part of the road network.
- The most popular networks are those with the least restrictions and the greatest connectivity. Thus the railways ousted the canals in the nineteenth century; the roads have almost ousted the railways today.

Computer networks are not so different at an abstract level from the other kinds mentioned above. The objects connected together are computers and computer terminals of all kinds. The paths connecting them together are copper cables in the simplest case (but much more complex possibilities exist). The "something" sent along these links consists of binary data, but of course these data may serve many different purposes according to the particular type of service concerned.

What are such networks good for? There are of course many possibilities, of which the following may be considered relevant in the High Energy Physics (HEP) field:

- Sending messages to people who are in another building, another university, another country, or another continent. Computer-based message systems (electronic mail) may often be more convenient, cheaper, or faster than other methods of human communication.
- Logging into a computer which is in another building, university, country, or continent. This may be more convenient, cheaper or faster than going there!
- Transmitting a data file, a program, a document, or a picture between two separate computers. This is certainly more convenient and faster than copying the information to and from a magnetic tape or floppy disc which must be sent by post, but it may well be more expensive.
- Running a job on a remote computer. If all you have is a PDP-11 and Rutherford has a Cray X-MP, it will be very attractive to run your job at Rutherford via the network.
On-line control or monitoring either of an experiment or of an accelerator today involves many mini-computers and/or micro-processors. These must exchange commands and responses in real time, and some of the above services (at least remote login and file transfer) will be needed within the control and monitoring network. I will not, however, cover this important area in these notes.

2. WHY IS IT SO DIFFICULT?

If you have had any contact with computer networks, in theory or in practice, you probably have the impression that this is a very complicated area. Why is this so?

- A first problem is that there is no such thing as a perfect transmission path. In good conditions, a Local Area Network (LAN) may corrupt one bit in $10^{12}$, as long as nobody touches anything. But degradation of circuits or connectors, or some kind of mechanical or electrical interference, can cause the error rate to climb dramatically for short or long periods. The situation is much worse for Wide Area Networks (WANs) used between cities. Instead of a simple copper wire, the transmission path may be very complex and error rates of one bit in $10^4$ are not unusual. There may also be brief breaks in the connection.

If we want to send a 10 kbyte file from Brussels to Vienna, that amounts to $10^5$ bits including some overhead. On a bad day, several bits will be corrupted. Obviously, the computer interfaces and the software for networks have to take account of this certainty that there will be errors.

- Misbehaviour by one or the other of the computers involved is also certain to occur from time to time. One machine may be overloaded and respond so slowly that the other machine gives up waiting. Or a machine being used as an intermediate switching point may crash, so that a new path must be used. Or there may simply be a minor error in the network software, producing effects just like a transmission error.

- The two computers may not be of the same make, or may be running different software (e.g. the IBMs at CERN and DESY may need to communicate, but one of them runs the MVS operating system and the other runs VM/CMS).

- The same links must be shared by many users at the same time, using various different services simultaneously, without interference or unfairness.

- As in any sophisticated technology, the specialists have invented their own shorthand jargon, which is normally confusing to the user.

The result of these and other facts of life is indeed that networks are complicated to design, implement and operate. Unfortunately, at least with the very general networks used in the HEP world, this complexity is often visible to the user, who is of course only interested in the high-level services and not in the nuts and bolts. Nevertheless, the following few chapters take a brief look at the nuts and bolts.

3. TRANSMISSION TECHNOLOGY

Because of the economics of cabling, the transmission technologies used in computer networks are in virtually all cases bit serial, i.e. the data bits are transmitted one after another along the same channel. This channel may be

- A very cheap twisted pair of copper wires such as a telephone circuit.
- A simple coaxial cable (as for a TV antenna).
- A high quality coaxial cable (often multiplexed between many users).
• A group of twisted pairs, with data and control bits flowing in both directions on separate pairs.
• An optical fibre (often multiplexed between many users).
• A microwave link.
• A satellite link.

A long distance link (WAN link) is normally rented from the national telecommunications monopolies (usually called "the PTTs"). It will consist of a concatenation of the various types of links mentioned above, in a way decided by the PTTs concerned. The quality of service will typically be that of telephone-quality twisted pairs, with all their risks of noise and short interruptions. One cannot simply send data bits down a link of this quality, and a modulation technique must be used. At each end of the link, a small box called a modem \(\text{(modulator/demodulator)}\) is connected to the PTT plug. This box accepts a stream of bits from the computer (or terminal), converts them to a modulated audio-frequency signal, and sends this signal down the link. The modem at the other end converts the audio signal back to a bit stream and delivers it to the remote computer. This whole process works simultaneously in both directions (\textit{full duplex working}).

The audio modulation, coupled with special techniques for detecting and correcting corrupted bits, permits useable (but imperfect) transmission at speeds normally between 300 bits per second (bps) and 19.2 kbps, according to the quality of PTT line and the type of modem used. Higher speeds (up to 64 kbps) require lines of better than audio quality, which the PTTs are today just able to supply internationally.

Satellite links normally end in a modem (although high speeds, above 64 kbps, require expensive modems). For very long distances, satellite links are normally cheaper than terrestrial leased lines, but they have the disadvantage of a 300 msec transmission delay which is very annoying for remote login use.

The connection between a modem and a computer uses a multi-pair cable following recommendations of the CCITT (the International Telegraph and Telephone Consultative Committee). The connectors and transmission speeds are standardised. But one must not forget that due to the intrinsic poor quality of telephone circuits, 
\textit{bit error rates of one in }10^5\text{ or one in }10^4\text{ are not unusual.}

The situation is healthier for LANs. Here connection distances are typically up to a kilometre, so cable can be of high quality and there are few if any intermediate connectors. Bit rates anywhere between 100 kbps and 100 Mbps are practicable, according to requirements and budgets. Error rates may be as good as one in \(10^{12}\), although it is safer to assume that the rate may increase to one in \(10^8\) under adverse conditions.

Many LAN technologies have been developed in the last ten years, but I will describe only one, normally known as Ethernet. This was originally designed by Xerox, and has been vigorously supported by Intel and Digital Equipment Corporation (DEC). It is also an American and international standard, with very slight modifications compared to the original industrial version.

Ethernet runs on a special type of yellow coaxial cable about one centimetre thick. It can also run for short distances on ordinary coaxial cable, when it is known as Thin Wire Ethernet or Cheapernet. Bits are transmitted at 10 Mbps without modulation. All stations (computers etc.) are connected to the same cable, just as side roads are connected to a highway. When a station wants to make a transmission, it "listens" to see if the cable is busy, and if not it starts transmitting. This is analogous to a car driver listening to see if the highway is free before pulling out of a side road. On both Ethernet and the road, this method can occasionally lead to collisions. Fortunately, a collision between two transmissions on Ethernet is not a disaster: both transmitters stop, and try again after a short random wait. This technique works very well as long as the load on the Ethernet does not exceed about 30% (i.e. not more than about 3 Mbps useful throughput).

Apart from error rates and transmission speeds, there is a third important difference between WAN and LAN transmission technology. WAN links are \textit{point-to-point}, i.e. they join computer A to computer B. LAN links are usually \textit{multi-drop}, i.e. computers X, Y, Z, etc. are all sharing the same cable. Thus any message sent by A on the WAN link can only go to B, whereas a message sent by X on the LAN must be specifically addressed to Y or Z etc. Also, if A is connected to B by a WAN link,
and B is similarly connected to C, messages from A to C must be sent to B and then re-transmitted to C. So although A does not need to specify a physical address to send a message, it must specify a logical address to distinguish messages meant for B from those for C.

In a network consisting of a mixture of LAN and WAN links, both physical and logical addresses must be handled.

On both LANs and WANs we can see that many users share the same cable (unlike a circuit-switching system such as the telephones, where a physical connection is reserved for a given pair of users throughout their conversation). The cable is shared by a technique called packet switching in which a sender must split the data into packets of a fixed maximum size (about 1500 bytes on Ethernet). Where a packet must pass through an intermediate node (such as B in the above example), this node must route the packet from the incoming line to the outgoing line according to its logical address. Packets from various sources are interleaved in order to share out the available bandwidth. In general these activities are hidden from the user.

A final aspect of transmission technology which I will mention is the computer interface. According to the speed and complexity involved, a computer interface to a network may cost between $200 and $50,000. It may be capable of moving one byte at a time to or from the network, or it may contain a 32-bit microprocessor completely handling complicated protocols (see next chapter). Thus it is very difficult to make any general statements about interfaces. However, it can be said with confidence that today, the limiting factor on the performance of LANs is the throughput of the computer interfaces. Achieving 100 kbyte/second throughput between two computers on a 10 Mbps Ethernet is considered to be very satisfactory performance. This problem is less acute for WANs where the transmission speed is typically not more than 9.6 kbps. Even so, driving such a line continuously at full speed with a cheap interface may be a heavy load on a minicomputer.

4. PROTOCOLS

The various difficulties and complexities mentioned in the last two chapters have to be overcome. This means that specific techniques and procedures must be defined for the use of each type of communication link, and for each type of service using it. These techniques and procedures must be in the form of a set of unambiguous rules; any given set of rules is commonly known as a protocol. Protocols are implemented either in hardware (electronics) or in software, and such an implementation is often also loosely referred to as a protocol.

The design of a protocol is a very fastidious intellectual exercise, since all possible eventualities must be catered for. To take a simple example, the connection of a terminal to a computer often employs the so-called "XON/XOFF" protocol. When one side sends an "XOFF" control character to the other, the other is supposed to stop sending any more characters until it receives an "XON" control character.

- What happens if both sides send "XOFF" simultaneously?
- What happens if an "XON" is corrupted in transmission?

Protocol designers must provide rules to cover every such awkward case, however unlikely it is to occur. Formal design techniques are now available — but nevertheless errors remain even in standardised and widely used protocols. Home-made protocols always contain errors.

The objectives of a set of protocols are in general:

- **at low level**, error detection and correction, flow control, multiplexing (sharing between users), establishing connections;

- **at high level**, managing a communication session, converting data formats, providing a user service.

Protocols exist to cover the following levels of abstraction:
1. **Physical**: Modems, Ethernet interfaces, etc. obey physical transmission rules including error detection by use of redundancy.

2. **Logical link**: Software or firmware drivers obey rules arranging for the error-free transmission of a packet of data bits from one computer to another along a single WAN or LAN link.

3. **Network level**: Software or firmware obeys rules arranging for a packet to be transmitted from one computer to another through a network of WAN and/or LAN links.

4. **Data transport**: Software or firmware obeys rules for a stream of data to be transmitted from one program to another. This is the lowest level of protocol which may directly interest a user.

5. **Session services**: Software may be available (not in all systems) obeying rules to facilitate a session of interaction between two programs or between a remote terminal and a program.

6. **Presentation services**: Software may be available (not in all systems) obeying rules to present data in an appropriate format at each end of a communication session. The classical case is that of a terminal using the ASCII character set, communicating with an IBM computer using the EBCDIC character set.

7. **Application services**: This is the important part for the user. Software must be available to provide the required high-level service (remote login, mail, file transfer, job transfer) according to agreed rules. These protocols are built on top of all the underlying layers, although some layers may be empty in cases where they are not needed.

An unfortunate fact of life is that there are many different protocols corresponding to each of the levels mentioned above. Also, some older protocols do not fit properly into the above classification. The following chapters attempt to make this situation more understandable.

### 5. ARCHITECTURES

As we have just seen, the user is mainly interested in so-called **Level 7 Services**, such as remote login. The physical network (level 1) consists of a collection of computers and of LAN and modem links of various kinds. The way in which the protocol layers between 1 and 7 are fitted together to provide a reasonably coherent total system is normally called an **architecture**.

One can distinguish two main types of architecture:

- **Manufacturer's architectures.** A given computer manufacturer such as DEC or IBM produces a coherent set of protocols and provides the management tools for fitting them together. Thus DEC support Digital Network Architecture (DNA) which allows DEC computers to be connected together (normally called DECnet). IBM support Systems Network Architecture (SNA) which allows IBM computers to be connected together. Many other manufacturers have similar (but incompatible) architectures.

- **Multi-vendor architectures.** There are a number of network architectures which are largely independent of any particular make of computer. An example is CERNET, the set of protocols developed at CERN ten years ago — available on IBM, DEC, Norsk Data computers among others. Another example is the TCP/IP family of protocols, originally specified by the American Department of Defense for their research network (ARPA.net), but now available on a wide variety of large and small computers.

In general, the protocols used in these different architectures are different (with very few exceptions, mainly at the physical level). Thus different architectures cannot talk to each other. Certain commercial products exist to get round this problem. This is done by arranging for one computer to **emulate** the protocols normally used on another — e.g. a Norsk Data computer running software which actually follows IBM protocols. Such a computer can be connected to an IBM — which "thinks" that it is really talking to another IBM. An emulator can be used as the basis for a **gateway**, which is a computer
connected to two or more different networks and translating between various different (but logically equivalent) protocols.

For example, CERN operates several different types of gateway at various protocol levels:

- **FRIGATE** is an Ethernet—tō—CERNET gateway operating at the physical address level (technically called a bridge). CERN has ten such bridges, some of which also offer a file transfer gateway between Ethernet and CERNET.
- **GIFT** is a file transfer gateway allowing file transfers between CERNET, DECnet and JANET (see below) despite their totally incompatible protocols.
- **MINT** is a set of electronic mail gateways allowing mail transfers between IBM, UNIX, DECnet and other networks.

### 6. OPEN SYSTEMS AND HEP POLICY

Networking is confusing and complicated, even though this presentation is a simplified one. Yet in reviewing the above chapters, it is hard to find a point at which complexity has been introduced which could have been avoided. *So the main problem for a network user is complexity and diversity.*

Although the large computer companies may not agree, everybody’s dream is of what is called *Open Systems Interconnection (OSI)*. This phrase means that any two computer systems, anywhere in the world, should be able to communicate using the same set of protocols. This is no longer entirely a dream. For ten years, international standardisation efforts have been in progress under the auspices of CCITT and ISO (the International Standards Organisation). More than 75 standards have been finished or are on the way under the banner of OSI. Some of these are pre-existing PTT standards brought up to date (e.g. the X.25 standard for public WAN services). Some of them are adapted industrial standards (e.g. ISO 8802/3, which is Ethernet). Some of them are completely new (e.g. ISO 8072, the Level 4 Transport Protocol, or CCITT X.400, the electronic mail protocol). Now, in 1986, implementations of some of these OSI protocols are becoming available on many makes of computer including DEC and IBM. Today, they run in parallel with older network architectures. In a few years, one may hope to see OSI displace the old solutions.

As a very diverse and international community, HEP has suffered greatly from the diversity of existing network architectures. A commercial company, or a normal government or military organisation, can standardise on a given make of computer or on a given multi-vendor architecture. HEP is a loose agglomeration of laboratories and universities and standardisation has proved difficult. A working group\(^1\) was set up some years ago to study the problem. Apart from proposing or encouraging many short-term solutions such as the GIFT gateway mentioned above, it has recommended that HEP adopts OSI protocols as soon as practicable. CERN is following the same policy. However, this is a rather theoretical policy for today’s users and I will now turn to what is really available.

### 7. THE MAJOR HEP NETWORKS

#### 7.1 Public X.25

X.25 is the name of the CCITT recommendation for public packet-switching networks, and it has been widely adopted. It specifies link and network level protocols, including procedures for establishing a connection (‘virtual circuit’) between two users.

---

\(^1\) Sub-Group 5 of the Data Processing Harmonisation Group set up by ECFA, the European Committee for Future Accelerators.
The PTTs almost all operate national public X.25 networks to which anyone can subscribe, as to telephone or telex. The national networks all have different names (e.g. Telepac in Switzerland, PSS in Britain) but underneath they are similar. There are international connections between them and a single international addressing scheme which is much more rational than international telephone codes. Thus, in the developed countries, we have a global public data network. It provides a level 3 service in terms of the list in the previous chapter, i.e. it is not a network providing user-level services. Thus high-level protocols must be provided at each end. For small institutes with low requirements, use of the X.25 network is economically interesting. Only large institutes (or rich companies) can justify the considerable costs of private leased lines. Even on these lines (of which there are at least fifteen coming into CERN), it is not unusual to use the X.25 protocol as a common basis.

The public X.25 service has the advantages of being fully supported by the PTTs, and of being cheap for low levels of usage. Its disadvantages are that it is slow and expensive for very heavy traffic, when leased lines become more interesting. There are however legal conditions attached to the use of leased lines, to protect the PTT monopolies in certain countries.

7.2 EARN and BITNET

Many sites in Europe are connected to the EARN network, and many sites in America are connected to BITNET. From a technical point of view, EARN and BITNET are the same network, the difference being an administrative one. The majority of the computers connected are IBMs and EARN/BITNET uses old-fashioned IBM protocols called RSCS (Remote Spooling Control System). However, many VAXen and some other types of computer are connected by the emulation technique.

EARN provides several basic services: file transfer, job transfer, electronic mail, and an unusual "real time messaging" facility allowing a sort of computerised telephone call through the terminal. The exact range and quality of services depends on the computer and operating system, and probably EARN is most suitable for IBM VM/CMS systems. Today, EARN is a very important network for HEP, although there are some question marks about its future after 1987 for financial and legal reasons. However, if all goes well, EARN will slowly migrate to OSI protocols.

7.3 The Physics DEcnets

It is hard to say whether there is one physics DEcnets or several, since the management is informal. However, several hundred DECs in the European and American physics institutes (including some 30 machines at CERN) are connected together by a mixture of Ethernets, leased WAN links, and connections through the PTT X.25 networks. DEcnets provides file transfer, electronic mail, and remote login between DECs. It comes very close to fitting the seven layer model described above, and DEC have announced that the next major release of DEcnets will include fully integrated OSI protocols.

DEcnets is undoubtedly the network architecture of choice if the main requirement is communication with DEC computers.

7.4 Triple X

This is not exactly a network, but it seems appropriate to include it here. Triple X is the common name for use of a set of three CCITT protocols called X.3, X.28 and X.29 which together specify how an ordinary ASCII terminal can connect to a remote site through the public X.25 network (or through a private leased line running X.25).

The normal procedure is that the user connects his terminal (directly or through a terminal switch) to a device known as a PAD (packet assembler – disassembler), which assembles the characters typed by the user into a packet and sends them off through the X.25 network. When a reply packet arrives from the far end, the PAD disassembles it into a sequence of characters which are then sent to the terminal.
At the computer end, a piece of software often called a reverse PAD carries out the inverse operation.

Since a PAD often serves to connect several terminals to the network it is sometimes called a terminal concentrator, although this phrase is more general.

### 7.5 The UNIX network

Thousands of computers running the UNIX operating system are connected to an international network called EUNET (in Europe) and USENET (in America). Like EARN, EUNET uses a rather simple protocol, called uucp (Unix–Unix Copy). It provides mail, news, and file transfer services and is very widely used by the computer science community. So far it has found little use for HEP, although computing specialists at CERN make effective use of it for keeping in touch with developments.

### 7.6 National Networks

The above networks are international in nature. In some countries, the academic community has set up a national network — or in some cases, more than one. The European countries that have not already taken this step are in general planning it, or running pilot projects. Some important national networks are:

- **JANET**: This is the British "Joint Academic Network" set up some years ago. JANET uses a set of protocols running on top of X.25 which are generally known as "the coloured books" because they are defined in a series of documents with various colours of cover. They have been implemented on a wide variety of computers in use in British universities and research institutes.

  Several computers at CERN are logically (but not administratively) part of JANET. The coloured book software is used quite widely on HEP computers outside Britain, including computers of the LEP experiment L3 ("LEP3NET"), and computers of the French physics laboratories ("PHYNET"). It can be used (like DECnet) over public or private X.25 circuits. Its main advantage is that it exists for both DEC and IBM (VM/CMS) systems; its main disadvantage is relatively weak support outside Britain.

- **DFN**: This is the German "Deutsche Forschungsnetz" which is now entering service. Since DFN plan a very early migration to OSI protocols, their initial set of interim protocols is unlikely to be of wide interest.

- **The rest of Europe**: Elsewhere in Europe there is a complex pattern of partial or pilot networks for academic use, and in most countries one finds plans for future OSI networks, all to be interlinked. There are in particular several pilot mail networks using a prototype implementation of the X.400 standard known as EAN. All these pilots and plans assume the use of the public X.25 network.

- **USA**: In the USA there is no tendency towards a single academic network. In addition to BITNET (EARN) and the US physics DECnet (known as PHYSNET), many physicists can access MFNet (set up for magnetic fusion energy research), ARPAnet (set up by the Department of Defense), and several other networks. There is a risk that progress towards OSI will be slower in the USA than in Europe, and that different options will be chosen.

What does all this mean for the poor HEP user? When you think you need to use a network you must ask a number of questions:

- I am at computer X on site A and I need to communicate with computer Y at site B. To which networks are X and Y connected? Do those networks offer the high level service I require?
• If X and Y are not on the same network, are there any gateways between their respective networks? Do these gateways offer the high level service I require?
• How much will it cost me or my boss? Do we have the money?
• After all that, do I still believe that a computer network is the answer to my problem? Is it really cheaper or more convenient than the alternatives?

You may find the answers to these questions (except the last one) in the documentation available at your computer site. It is unfortunately much more likely, however, that you will have to consult specialists, or colleagues who have met similar situations before. The following chapters assume that you have obtained satisfactory answers and that you have chosen the network to be used.

8. ELECTRONIC MAIL: GETTING STARTED

8.1 Which system?

There is no general answer to this question; as with the choice of networks in general, you must ask for advice from colleagues with a similar range of needs. At CERN, a particular mail system has been recommended for each operating system, but there is no automatic reason for this choice to be valid at other sites.

Thus there is no general recipe for getting started. Here are some features of most mail systems which are essential or highly useful. It is worth learning how the mail system you use implements these features.

• Sending and reading messages. These are of course the fundamental facilities.
• Saving addresses. It is difficult to discover electronic mail addresses. Once you have worked out somebody’s address, you need to save it. This is normally possible using an alias facility, sometimes known as a nickname. Some systems also provide a directory service.
• Saving messages. It is important to save messages that you have sent or received for future reference. This is normally possible using a folder or file system.
• Replying and Forwarding. Many systems have a reply and a forward command, allowing easy handling of messages without retyping too much. Beware, however, that messages which have come through gateways may cause trouble with a reply command, since gateways often have asymmetric behaviour.
• Copies and distribution lists. Many systems allow you to send a message to multiple destinations, or to send copies to secondary destinations. Distribution lists provided by the mail system are very useful when organising meetings etc.

8.2 Addressing

There are several common addressing schemes for electronic mail, and when a message has to be routed from one network to another through a gateway, you may need to use an address which is a mixture of several formats. Some common formats are:

• DECaet: HOST::USER where HOST is a computer name and USER is the name of the user’s account on that computer. In complicated DECnet networks, you may have to use the format HOST1::HOST2::USER where HOST1 is an intermediate computer between you and HOST.

• EARN: USER AT HOST (or in some contexts USER@HOST ). Host names are unique throughout EARN and BITNET.

• CERN standard format: At CERN, wherever possible mail addresses take the format user@host.domain where domain is either a single name (e.g. cern, the default), or compound (e.g. rl.ac.uk, for Rutherford Laboratory – Academic Community – United Kingdom).
Thus valid address formats include:

user@host.cern
user@host.earn
user@host.decnet
user@host.rl.ac.uk

This format obeys a de facto standard widely used in mail networks, since the international standards in this area are not yet clear. Many mail networks in Europe and the USA can be accessed from CERN using this format.

- **JANET and other Coloured Book hosts**: user@domain.host where domain is a composite form such as uk.ac.rl (i.e. United Kingdom – Academic Community – Rutherford Laboratory).

 Note the inverted order compared to the de facto standard! This inversion is a very common source of error.

- **UNIX network**: host1user or host1/host2/.../host1user

If the local computer knows a route to the remote host, the first form can be used. Otherwise the user has to specify a complete route as in the second case.

Note that in UNIX computers, upper case and lower case letters are distinguished, so FRED, fred and Fred are three different users.

- **Mixed addresses**: Two important cases (there may be others) of mixed address formats can arise when mail must be routed through gateways:

1. *Two "@" signs needed.* The first "@" is replaced by a "%", i.e. user%host@gateway

Thus a JANET user can address a CERN user as user%CERN.host@UK.AC.UCL.CS using an experimental gateway in London. Note the use of JANET address ordering.

2. *Mail leaving DECnet.* A "foreign" address format will be used, i.e. VXGATE::"user@host.domain" (VXGATE is an imaginary gateway VAX).

### 8.3 Pitfalls

Electronic mail is not 100% reliable today. Messages may be lost without warning (especially when passing through gateways). You should always build in a check for important messages (e.g. "Please acknowledge before next Tuesday.")

Some people never (or hardly ever) read their electronic mail despite having a valid account.

Addresses change very often. It is a constant job to keep your file of aliases or nicknames up to date.

If possible, use only one mail system. It simplifies your life and that of your colleagues. Some systems have an 'auto-forward' facility, so that mail for several accounts can be routed to your preferred system.

### 9. REMOTE LOGIN: GETTING STARTED

#### 9.1 DECnet and TCP/IP

We will consider these first because they are very simple. If you are logged in to a given DECnet machine, and if your account has the necessary privilege, you can connect through the network to another DECnet machine by typing
where VXABC is the other machine (a VAX in this example). You will then get the normal login prompt from the remote machine. Of course you must also have an account on that machine.

Note that the performance of SET HOST is acceptable on a LAN, but that if a WAN link is involved you will obviously notice a slow-down.

The TCP/IP remote login system called Telnet is equally simple. On a UNIX or other TCP/IP-based system, you simply type

telnet abc

to log in to computer ‘abc’.

9.2 Triple X

The recipe for calling the Triple X service varies so widely that little can be said except ‘ask your colleagues’. At CERN, the service may be called by selecting class XXX on the general terminal switch (INDEX), and you will need a budget group and a password. You must also know the multi-digit X.25 address of the remote system.

Remember that in Triple X, your terminal talks to a PAD, the PAD normally talks to the international X.25 network, and at the other end there is a reverse PAD talking to the computer. In some cases, life may be even more complex. For example, when you call CERN from outside, you have a choice of X.25 addresses. Some of these connect to built-in PADs in certain computers such as the IBM VM/CMS system. One of them connects to the general terminal switching system giving a choice of more than 100 computers. Some of them connect to computers with no PAD, so Triple X login is impossible on those addresses.

Another complexity with Triple X is that the standards contain a number of options (PAD parameters) connected with the handling of special characters etc. The particular set of parameters in use is called the profile and if the profiles of the two PADs are incompatible there can be problems. It is very tricky to set up profiles that will work in all cases, especially for an international site like CERN, so yet again you must consult with colleagues (or even try looking in the documentation).

Once you have succeeded in logging in via Triple X (it is sometimes necessary to try several times) you will certainly notice a slow response time compared to local login. This is particularly true when running internationally, since the international links in the X.25 network tend to be bottlenecks. There is also a non-negligible risk of apparently random disconnections in international traffic. Use of screen-oriented editors tends to be frustrating in these conditions (but may be acceptable when using Triple X on leased lines).

10. FILE AND JOB TRANSFER: GETTING STARTED

In this area again, almost no general statements can be made. Note that some file transfer systems (e.g. DECnet) rely on a complete path being open, since files are transferred block by block in real time. This has the advantage that immediate notification of errors is possible. Other systems (e.g. EARN) use the ‘store and forward’ principle, meaning that files are stored on disc at each intermediate node in the network. This has the advantage that the path may be temporarily broken without preventing the transfer, but the user may not be notified of errors (as with mail, which is normally ‘store and forward’).

10.1 DECnet

Copying a file through the network is achieved by the command
where VXSRC, VXDEST are the source and destination computers and SRCETYPE, DESTETYPE are the source and destination files. In some cases, a password must be supplied.

A job can be run remotely by copying the batch file and using SET HOST to log in at the far end and run it.

10.2 EARN

The exact details vary between operating systems. On IBM VM/CMS, files are sent by one user to another using the standard command SENDFILE. On-line help is available for this command, and you will need to know the user and host name of the receiving user. As in the case of mail, the file will arrive in the remote user's 'reader' (simulated card reader) if (s)he is also on VM/CMS. In the case of other operating systems, the file will just be placed in the user's directory.

To submit a batch job through the network, you must first create a file containing the job control statements (JCL) and any necessary data. On VM/CMS you must then set up your 'punch' (simulated card punch) using the TAG and SPOOL commands so that simulated punched card decks are sent to the computer where you want to run the job. You must then use the PUNCH command to send the JCL file off. If all goes well the output will be routed back to you.

In other cases, you may be able to use a command such as RUN ON host but for this and other details please consult local documentation and specialists.

10.3 TCP/IP

TCP/IP actually has two file transfer protocols, one called FTP and the other called TFTP (trivial FTP!). Both are very easy to use, with get and put commands to transfer complete files, and several ancillary commands to connect to the remote computer, supply a user name and password, etc.

10.4 GIFT

CERN operates a file transfer gateway called GIFT (General Internet File Transfer). At present this allows file transfers between CERNET, JANET and DECnet, with more possibilities planned. If you are on one of these networks and need to move a file to one of the others, GIFT may solve your problem (but only if one of the computers concerned is at CERN). Use of this is fairly simple if you know the file transfer method on your own network and the file naming syntax on the remote network. A comprehensive set of guides is available.

11. FUTURE PLANS AND PROSPECTS

The state of networking today in the HEP world (and in the world in general) is confused and confusing. What are the prospects for the future?

1. Over the next five years, the OSI protocols will arrive in force. Most computer manufacturers of any importance have already announced plans (including DEC and IBM). There are various moves to exploit OSI protocols in the academic and research communities, including HEP, both on a national and international basis. If these moves succeed, we may hope that in a few years, the world is no longer split up into separate networks such as EARN, DECnet, etc., but that there is one general network used by all of HEP.
2. However, a wise man has said that there are always four standards: the old standard, the current standard, the new standard, and the non-standard standard. Thus one must be aware that although OSI should reduce the fragmentation of networks, advances in technology will mean that things will continue to change and that gateways between old, new and interim protocols will always be needed.

3. There is less chance that user interfaces to networks will be standardised. So even when everybody is using OSI protocols, you will probably still need to learn different recipes in order to access the network from different systems.

4. I have not emphasised problems of performance in these notes. However, there are of course performance limits in current networks. For LANs, as mentioned earlier, the limit is normally the throughput of the computer interface and of the protocol software. As micro-electronics gets cheaper and cheaper, one may hope for this limit to be pushed back, but it is likely to remain a limiting factor for applications such as moving graphics images between a super-computer and a workstation.

For WANs, the limit is not so much the speed of a long-distance link as its cost. As experiments at CERN and elsewhere have shown, it is not difficult to exploit links running at 1 Mbps or more over long distances. Indeed we may expect that much faster links, say 140 Mbps, will be available in Europe in a few years via international optical fibre connections. Digital links up to 2 Mbps via satellites may also become generally available. At the same time the PTTs will be offering so-called Integrated Services Digital Networks (ISDN) giving ready access to long-distance 64 kbps (and faster) digital circuits which can be switched through in a fraction of a second. Compared to today's 9600 bps leased lines and slow X.25 packet-switching networks, this ought to be a revolution in performance. Fortunately the ISDN standards are reasonably well integrated with the OSI standards, so the transition to ISDN should not affect users too much. But will the PTTs set a reasonable tariff for all these new services, or will HEP stay at 9600 bps for lack of money?

Acknowledgements: The mistakes in these notes are all my own. However, I am very grateful to the following people for their invaluable comments: Francois Fluckiger, Mervyn Hine, Ben Segal.

12. APPENDIX: SOURCES OF FURTHER INFORMATION

A good general book is Computer Networks by A.S.Tanenbaum, Prentice-Hall, 1981, although it is now slightly out of date on OSI protocols.

Please consult the people responsible at your own institute or university to obtain the documentation for the networks available to you.

At CERN, the following documents may be of interest (among others):

A user guide to electronic mail services at CERN, DD/US/6 (from the Computer Science Library self-service, or FIND MAILGUIDE on VM/CMS).

EARN/BITNET user's guide, by Olivier H. Martin (TELL NETSERV AT CEARN GET CERN RULES on VM/CMS).

GIFT manual & GIFT pocket guides, by G.Heiman and colleagues.

EXTASE pocket guide, (external Triple X service), by R.P.O'Brien and colleagues (HELP WRITEUP EXTASE on Wylbur, FIND on VM/CMS).

CERNET user guide, DD/US/33, (from the Computer Science Library self-service, or HELP WRITEUP CERNET on Wylbur, FIND on VM/CMS.)

TCP/IP Networking at CERN, by B.Segal and G.Lee.
COMPUTER ARCHITECTURES FOR HIGH ENERGY PHYSICS

Richard P. Mount
California Institute of Technology, Pasadena, California, USA

ABSTRACT

The increasing cost of HEP computing leads us to seek more cost-effective sources of computing power. Vector, deep-pipeline, and parallel computer architectures are reviewed, and the results of performance tests are given, leading to a cost/performance/completeness analysis for a range of available systems. The software needed to make efficient use of vector and parallel computers is then examined in more detail.

1. INTRODUCTION

1.1 Why am I here at all?

I am a physicist, and as such, I am more accustomed to being a user of computer systems than a buyer or creator of them. My interest in the computing needs of HEP, and of peculiar ways in which these needs might be met, stems from my involvement, from its earliest stages, in the L3 experiment at the LEP accelerator.

LEP is the 27km circumference electron-positron colliding-beam accelerator now being constructed in a circular tunnel straddling the Swiss-French border near Geneva. When the construction is complete in early 1989, four large ‘experiments’ will measure the products of the electron-positron collisions occurring at four equally spaced positions around the LEP ring. L3 is physically the largest experiment, and arguably the most computationally demanding due to its many very high precision detector systems.

A few numbers may help to impress and, perhaps, even inform. The L3 experiment will be assembled in a huge artificial cavern, 50m below the surface. Externally, L3 will appear to be a cylindrical magnet, 14m long and 17m in diameter, with a tube, 32m long and 4.4m in diameter running along the axis of the magnet. The magnet, containing 6000 tons of steel, and 1000 tons of aluminium coil will have a free bore of nearly 12m diameter by 12m length and a field of 5000 gauss. The tube will support the 700 tons of instrumented detectors which cover almost all the solid angle around the ‘interaction point’ at the centre of the structure.

Particles emerging from the central interaction point will be measured by tracking chambers and by calorimetry (total absorption in energy-sensitive devices). Most of the L3 detector systems are breaking new ground in the precision which will be attained. For example, the massive ‘muon chamber’ system, which fills the magnetised volume outside the support tube, will measure the position of particle tracks to better than 30 microns. Such precision has only become possible by resorting to ambitious and, naturally, expensive, engineering techniques.

It is considered foolish ‘to spoil a good ship for a ha’p’orth of tar’. In this context the ‘tar’ is the provision of computing resources which will enable us to realise the potential of the L3 experiment, and the ‘halfpennyworth’ may be several million dollars worth. From the very beginning, we have worried about the computing resources we would need and how they might be provided. Here is the chronology of the L3 computing plans:
Based on experience with earlier, smaller, experiments, we estimated that we would need at least 50 VAX 11/780 equivalents on the CERN site for rapid off-line analysis of L3 data. (I will explain later why this is just a minimum). CERN planning, based on available finance and not on expected need, put the likely CERN contribution to L3 at about 12 VAX equivalents.

1982-1986

L3 searched for ways to provide the missing computing power. Firstly by trying to get as much finance as possible from our funding body (in this case the US DoE). Second by studying cost-effective architectures and benchmarking systems and components whenever possible.

1986-1987

A 'Request for Proposals for an Integrated Computer System for the L3 Collaboration', has recently been issued. Proposals are sought for a system comprising a Host plus attached processors which L3 will develop into a flexible parallel processing system. Offers are expected before the end of 1986 and installation will start in June 1987.

As a result of these activities, several L3 physicists have acquired an unexpected experience and understanding of computing architectures, and are frequently invited to give lectures such as these.

1.2 What is HEP Off-line Computing?

High energy physicists are greedy consumers of computing power. However, HEP cannot be satisfied by MIPS alone; for most tasks a full configuration involving expensive peripherals is a necessity. To explain and illustrate this I will outline the activities typical of HEP computing.

Software Development

In most HEP experiments, the majority of the code used to process and analyse the data is written specially for the experiment. L3 will have a software base of between 300,000 and 500,000 lines of FORTRAN, of which 80% will be written by L3 members. The computers used for software development must support many users (30 to 100 for L3). Typically the users develop and test code using Edit/Compile/Link/Run cycles requiring both good interactive response and substantial CPU power. Productivity is improved by making plenty of disk space available to each user, and by having good printing facilities. Tape usage is not heavy, but it is needed for serious tests of software approaching its final state.

Production Processing

Production processing involves using the data from the experiment to reconstruct particle tracks and energies. It is also necessary to generate a large volume of simulated data by Monte-Carlo techniques, and pass this through the same reconstruction program.
All production processing requires massive CPU power and a large I/O volume. However, the I/O to CPU ratio is never very high. A 50 VAX equivalent system might perform I/O at a rate below 100k bytes per second for production processing. Monte-Carlo event generation, due to its high CPU requirements, will perform an order of magnitude less I/O.

HEP data do not come in a continuous stream, but in the form of distinct entities called 'events' which contain all the data relating to a single collision between (in LEP) an electron and a positron. Events are independent, and may in principle be processed in any order, or in parallel.

Physics Analysis

Physics analysis starts when the routine production processing is finished, and uses all the facilities at our disposal to reduce a massive amount of data to (we hope) startling revelations of the new physics accessible to our experiment. Since we don't know what new things we will discover, we cannot write all our analysis tools in advance, and much software development activity is included in 'physics analysis'. In addition, to manipulate the massive data samples in imaginative ways, physicists need large CPU and I/O resources available on demand. As in production processing, the data are still in the form of events, which can, in principle, be processed in any order.

Naturally, as scientists, high energy physicists spend most of their computing power on massive floating point calculations......or do they? Figure 1 shows a randomly selected page of code taken from the GEANT program, which is a component in the L3 Monte-Carlo event simulator. Although my example (it really was chosen randomly) is perhaps a little extreme, more systematic analysis shows that floating point computations do not dominate HEP code, and that 'IF's and subroutine calls are very frequent. Vector arithmetic using vectors and matrices larger than 4 x 4 is uncommon even within floating point code.

1.3 Where Does the Money Go?

Computing systems which support the full range of HEP computing activities must have many peripherals. If you get a manufacturer's price list, and calculate the cost of a complete HEP computing centre, you will find that about 50% of your money is required for peripherals.

Looking at this in another way, even if you can reduce the cost of CPU power to near zero, with the aid of a bag of chips and a soldering iron, you still need a lot of money to set up an HEP computing centre.

In reality, the potential savings from cheap CPU power are even lower. Although it is realistic to do a lot of 'production processing' on cheap, home-made processors, it is very difficult to move highly interactive or peripheral-intensive work off commercial processors. Production processing uses 60 to 70% of the CPU power, but the remaining interactive and I/O intensive activities use at least 60 to 70% of the money.

Our search for new architectures for HEP must keep these cold realities in sight. However, although
smart computing ideas are unlikely to save much money, they may make feasible smart physics ideas, and thus greatly increase the real productivity of high energy physics experiments.

1.4 HEP CPU Needs and Conventional Solutions

Nobody need journey to a remote corner of the Netherlands to learn that HEP computing needs are increasing, whereas the cost of a unit of computing power is decreasing. However, if I can make some more quantitative statements, the travel might start to be vaguely justified.

HEP computing needs have continued to increase for many reasons. Here are some of them:

- Higher energy collisions generating more and more particles,
- Larger detectors (to contain higher energy particles),
• Fancier electronics, particularly ‘flash ADC’s’,

• More simulated data needed to match high-statistics, high-precision real data.

I will pick on one of these reasons for further explanation. In the old days (all of ten years ago) when we built a ‘wire-chamber’ we would send the signal from each wire to very expensive electronics which would set a single bit if a particle passed near the wire. These days, if we build a similar wire-chamber we will probably send the signal to a ‘flash-ADC’ which digitises the signal on the wire to a precision of 6 or 8 bits every few nanoseconds. Instead of one bit, we now have thousands.

The rapid increase in needed computing power might not be a major problem if the decreasing cost of computing always made it possible for us to buy all the power we needed. Sadly, computers are not getting cheap fast enough. Figure 2 shows both the evolution of computing costs and the evolution of experimental needs. The computing costs in $/MIPS* (as quoted in ‘Datamation’) are shown for IBM mainframes over the last 20 years. Computing needs of HEP experiments are not reviewed in Datamation, so I drew on my own experience and estimated the MIPS per physicist for the five experiments in which I have had a substantial involvement. It seems clear from this figure that conventional computers are indeed not getting cheap fast enough. For those who wonder whether the experiments in which I work are perhaps weird and atypical, Fig. 3 compares the slope of the MIPS per physicist in Fig. 2 with estimates of computing needs of the US national laboratories given in a recent report9 by a HEPAP subpanel. The long term extrapolation from my experience seems, if anything, low.

\begin{center}
\begin{tikzpicture}
\begin{axis}[
    title={The Decreasing Cost of IBM Mainframes Compared with the Increasing Needs of High Energy Physicists},
    xlabel={Year},
    ylabel={MIPS per High Energy Physicist},
    ylabel near ticks,
    xlabel near ticks,
    xmin=1965, xmax=1990,
    ymin=0.01, ymax=10.0,
    ytick={10.0, 1.0, 0.1, 0.01},
    yticklabels={10.0M, 1.0M, 0.1M, 0.01},
    legend pos=north west,
]
\addplot[only marks, mark options={solid}, mark=*]
coordinates {
(1965, 10.0) (1970, 1.0) (1975, 0.1) (1980, 0.01) (1985, 0.01) (1990, 0.01)
};
\end{axis}
\end{tikzpicture}
\end{center}

* I do agree that the MIP is a ‘Meaningless Indicator of Performance’ under most circumstances, but in the case of comparison between IBM machines (mainly systems 360/370) it is not too distorted, at least on a logarithmic plot.
The price of IBM mainframes was about the most relevant quantity I could have plotted in Fig. 2. The conventional solution to HEP computing needs have always been to go out and buy the fastest scalar processor you could afford. This was quite a good idea, since as we have already begun to see, HEP code is really quite well optimised for conventional scalar machines.

Only the spectre of a total insufficiency of computing resources has led us to examine more cost-effective ways of doing our computing. Saving money has, of course, never been the prime motivation, but the prospect of getting vastly more computing power for very little more money proves very exciting. Many HEP experiments have been seriously compute-bound (I won't give any names here):

- data analysed once only
- insufficient simulated data
- mistakes not allowed (to be admitted)
- new ideas for analysis not encouraged.

Removing these restrictions can increase the quality of physics results just as surely as building a better detector.

*Figure 3  Computing Needs of US National Laboratories*
2. THE FUNDAMENTAL PROBLEM

The fundamental problem facing all users of massive computing power is that the CDC 7600 is still a fast machine. Let me explain this cryptic statement.

CERN installed a CDC 7600 in 1972. It was the fastest computer in existence. When it was switched off, in August 1984, it was still one of the fastest scalar CPU's available. Even today, the fastest scalar CPU's only beat the CDC 7600 by a factor of two.

This story is partly a tribute to the genius of Seymour Cray, the designer of the CDC 7600, but it also illustrates how hard it has become to make faster scalar CPU's. Construction and maintenance costs continue to fall by about a factor of 2 every 4 years, but the cycle times of the fastest CPU's now decrease very slowly.

Thus users of large amounts of CPU power cannot continue to work exactly as they do now, using one or two powerful CPU's to meet all their needs. As computing requirements continue to rise, more and more applications will have to use hardware working in parallel, regardless of financial constraints.

3. ARCHITECTURAL REVIEW

This review of computer architectures will be pragmatic and simple enough for even HEP experimentalists to understand.

All the architectures I will describe are attempts to solve 'The Fundamental Problem' by performing many operations in parallel. The technology of the machines is the same as that used in fast scalar computers.

3.1 Vector Computers (e.g CRAY, CYBER 205)

The so-called vector computers achieve their speed by performing many identical operations (almost) simultaneously. Since this is typical of what is required in vector and matrix arithmetic, the machines have long been named 'vector computers'.

Figure 4 shows the time relationship between successive multiply instructions on a vector machine. Although a multiply takes typically 7 cycles, it is possible to initiate a multiply every machine cycle. Provided that the operands are made available in the appropriate 'vector registers' it takes just 63+7 cycles to perform 64 multiply operations.

\[ \text{multiply} \]
\[ \text{multiply} \]
\[ \text{multiply} \]
\[ \text{multiply} \]
\[ \text{multiply} \]

machine cycles

\text{Figure 4} \quad \text{Example of Successive Instructions on a Vector Computer}
These machines are expensive (in the 10M$ to 20M$ range), but they can be very cost-effective for programs which perform many identical operations in succession.

For comparison with other architectures I note some important features of vector computers:

- Single instruction stream.
- Parallelism invisible to the programmer.
- HEP code will run correctly on these machines, but making it efficient (known as 'vectorising') usually means re-writing it.
- Program speed-up depends mainly on how much 'unvectorised' execution remains.

3.2 Deep-Pipeline Computers (e.g. FPS 164, 264, 364)

I consider it much more 'natural' to build a computer which can perform many different operations (almost) simultaneously than to concentrate entirely on repetitions of identical operations. The FPS deep-pipeline computers have ten complete execution units enabling up to ten operations to be in progress at the same time.

Figure 5 shows the time relationship between successive instructions on an FPS computer. Obviously it is only possible to initiate an operation if it does not require the results of any operations still in progress.

```
multiply

divide

add

multiply

divide

machine cycles
```

*Figure 5*  Example of Successive Instructions on a Deep-pipeline Computer

FPS computers are cost-effective for programs which perform a lot of (not necessarily vector) arithmetic. Code with many conditional branches (see Fig. 1) executes inefficiently.

Once again I note some important features:

- Single instruction stream.
- Parallelism invisible to the programmer.
- Typical HEP code can keep two or three of the ten execution units busy.

3.3 Tightly-Coupled Parallel Computers (e.g. ELXSI 6400, Alliant FX/8)

Parallel computers execute several instruction streams simultaneously. Tight coupling normally means memory sharing, and overheads can often be kept low enough so that short sections of code can
execute in parallel quite efficiently. The conceptual structure of a tightly-coupled system is shown in Fig. 6.

![Diagram of shared memory with local memory and processors](image)

**Figure 6** The Conceptual Structure of a Tightly-coupled Parallel Computer

High bandwidth connections to shared memory are costly and work only over limited distances. Hence these machines are not necessarily cheap or infinitely expandable.

You may have noticed that the hardware of a tightly-coupled parallel computer looks just the same as that of a traditional multi-CPU mainframe such as an IBM 3084Q. The distinction is largely a matter of software, and recent developments have made it possible to use the 3080 and 3090 series computers for parallel processing. This can make single tasks execute up to four times faster. It does not, of course, make the computers any cheaper.

The important features of tightly-coupled parallel computers are:

- Single instruction stream parallelised by a special compiler,
  
  or
  
  - Parallel instruction streams controlled by the programmer. (Can get very tricky unless the memory sharing is disciplined.)
  
  - Process-process communications overhead is negligible.
  
  - Parallelising HEP code is best done by hand.
  
  - The processors can have vector or pipeline features, (the CRAY-XMP can be used as a 4-processor tightly coupled parallel system).

In my OPINION, tightly-coupled systems with many processors are the computers of the future. Although the tight coupling costs money and is not essential for most HEP event processing, it gives the machines a wide enough range of potential applications to make commercial success a real probability.

### 3.4 Loosely-Coupled Parallel Computers (e.g. Clementi Machine, Emulator Farm)

In a loosely-coupled system, inter-processor communication is by I/O only with maximum speeds in the region of a few megabytes per second. To make parallel processing efficient, relatively large sections of code must run in parallel on each processor; such large parallel 'subprograms' may not be recognisable by a parallelising compiler.
Figure 7 shows the conceptual structure of a loosely-coupled system. The driving principle in the design is usually the minimisation of cost. This leads naturally to relatively cheap attached processors with little or no ability to drive peripherals, coupled to a conventional 'host' mainframe system which provides the necessary disks, tapes and terminals, but little CPU power.

![Diagram of a loosely-coupled parallel computer]

*Figure 7 The Conceptual Structure of a Loosely-coupled Parallel Computer*

If the application requires little host-AP communications, AP's can be added almost indefinitely. A loosely-coupled system can be constructed by any do-it-yourself enthusiast with a little bit of money and a lot of enthusiasm. It is possible for the host and the AP's to have different instruction sets, data representations, and operating systems (if any). Making the whole thing work is 'only a matter of software'.

The important features of loosely-coupled systems are:

- Parallel instruction streams generated by the programmer. The instruction streams communicate only by I/O.
- Processor-host communications overhead is significant; don't try to parallelise small blocks of code.
- Parallelising HEP event-processing code is easy but the communications overheads limit the range of applicability.

In my OPINION, loosely-coupled systems are a good short-term solution for HEP provided that we use cheap AP's and beware the overheads.

4. BENCHMARKS AND ANECDOTES

Most of the information in this section has been gathered by myself and other members of L3 in the course of our search for a powerful, affordable, computer system.

4.1 Vector Computers (CRAY 1, CRAY X-MP, unmodified code)

The L3 collaboration has not seriously considered the purchase of a CRAY X-MP, so I can only report other people's benchmarks. The results obtained by the OPAL collaboration are typical. The
benchmark program was an 'unvectorised' shower Monte-Carlo. The only modification made for the benefit of the vector machines was to replace calls to matrix multiplying subroutines by in-line FORTRAN code.

Table 1 shows the CRAY performance in comparison with a conventional (IBM compatible) mainframe. The CRAY performance is close to that of a scalar CPU with the same cycle time, showing that the compiler was almost unable to use the vector hardware when faced with unmodified HEP code.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time</th>
<th>VAX 11/780/FPA Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray X-MP (on one CPU)</td>
<td>33 seconds</td>
<td>25</td>
</tr>
<tr>
<td>Cray 1</td>
<td>47 seconds</td>
<td>17</td>
</tr>
<tr>
<td>Siemens 7890 (on 1 CPU; scalar machine for comparison)</td>
<td>32 seconds</td>
<td>26</td>
</tr>
</tbody>
</table>

4.2 Vector Computers (Cyber 205, re-written code)

If we are prepared, not just to re-write, but also to re-think, tracking and reconstruction algorithms, execution speeds on vector machines can increase dramatically. A group at Florida State has re-thought a track-finding algorithm within the reconstruction code for Fermilab experiment E-711.9

The traditional way to find projected tracks in a wire-chamber pack is something like this:

- for each 'hit' in the first plane of the pack,
- and for each hit in the last plane in the pack,
- define a 'road' running from the first plane to the last plane,
- look in all the intermediate planes and select the hits lying within the road.

The totally different approach which was devised to take advantage of the large memory and the vector hardware of the Cyber 205 was:

- pre-generate all 89784 possible projected tracks in the chamber pack. Store all the projected track hit-patterns in memory.
- compare data with all the projected tracks.

A speed comparison between the original code running on a VAX, and the re-written code running on a Cyber 205 is shown in Table 2. For normal scalar HEP code, the expected Cyber/VAX speed-up would be about 20. Accordingly it is estimated that re-thinking the code gave a further speed-up of a factor of 10.
Table 2

Performance of a vector computer: re-written code

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time</th>
<th>VAX 11/780/FPA Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYBER 205</td>
<td>7.7 msec</td>
<td>208</td>
</tr>
<tr>
<td><em>(re-written code)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAX 11/780</td>
<td>1.6 seconds</td>
<td>1</td>
</tr>
<tr>
<td><em>(original code)</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I will discuss the merits of re-thinking code later. For the moment we should simply note that the large speed-up was achieved on a track-finding kernel, and not on a whole program, and that the identical approach would have been a disaster on a CRAY 1 or X-MP, both of which have very limited memories.

4.3 Deep-Pipeline Attached Processor (FPS 164)

The FPS 164 (and 364, 264) are 64-bit computers designed to be used as attached processors. They feature:

- 10 pipelined execution units.
- FPS-specific data format and instruction set.
- Good cross-compilation and cross-linking support, (at least on the IBM host system we tested).
- Good software and hardware support for automatic data translation provided you don’t use EQUIVALENCE.

In HEP Fortran code, it is normal to use EQUIVALENCE to allow the mixing of Real and Integer data-types in the same FORTRAN array. This gives the FPS software no chance of setting up the correct format conversions when data are to be transferred.

Fortunately, although the true data-type cannot be determined by the Fortran compiler, in well organised HEP programs the data-types are available so that the data can be translated to a machine-independent format when it has to be output. The code used to write and read magnetic tapes can also be used to ‘write’ to and ‘read’ from attached processors like the FPS 164.

L3 benchmarked the FPS 164 as a component of the ‘Clementi Machine’.\(^7\) The performance is shown in Table 3. Differences of more than an order of magnitude were found between the extremes of the ‘EMC DECODING’ package and ‘WIRCHA’ using the FPS Mathematical Library.

The ‘EMC DECODING’ unpacks data written by a 16-bit mini-computer. All the arithmetic is integer, there is a lot of bit manipulation and many conditional branches. It would be hard to find any code less suited to the 64-bit pipelined FPS.

The ‘WIRCHA’ program calculates the electric field in a wire-chamber particle detector. Most of the CPU-time is used to invert a 250 x 250 matrix, and performing this matrix inversion with a specially optimised FPS subroutine gave a very impressive result.

Between these extremes, the ‘GHEISHA’ hadronic shower simulator, and the ‘LUND’ electron-positron collision simulator are more typical of the bulk of HEP code.
Table 3

Performance of the FPS 164

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>Type of Code</th>
<th>VAX 11/780/FPA Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMC DECODING</td>
<td>Integer arithmetic, bit manipulation, many 'IF's.</td>
<td>2</td>
</tr>
<tr>
<td>GHEISHA</td>
<td>Single precision arithmetic, many 'IF's.</td>
<td>3</td>
</tr>
<tr>
<td>LUND</td>
<td>Some double precision arithmetic, fewer 'IF's.</td>
<td>6</td>
</tr>
<tr>
<td>WIRCHA</td>
<td>Double precision arithmetic, including $250 \times 250$ matrix inversion.</td>
<td>9</td>
</tr>
<tr>
<td>WIRCHA</td>
<td>As above, but using the FPS Mathematical Library.</td>
<td>26</td>
</tr>
</tbody>
</table>

4.4 Scalar Attached Processor (3081/E Emulator)

Emulators are 'home made' CPU's that run IBM object code translated 'on the fly', or in a previous step, into emulator microcode. Although commercial attached processors have become increasingly attractive in the 6 or 8 years since the emulators were first conceived, emulators remain one of the cheapest and easiest ways to add CPU power in an IBM computing environment.

The 3081/E has been developed in a joint CERN-SLAC project. Several processors are now in active service, for example in an emulator 'farm' run by the UA1 collaboration at Harvard, and in a farm run by CERN, L3 and UA1 at CERN.

The features of the 3081/E are:

- Limited pipelining, as in most modern 'scalar' processors. For example, the multiply and divide units can be in use at the same time.
- IBM data format.
- 64-bit hardware.
- Runs translated IBM object code. The translator re-arranges instructions to optimise pipelining.
- Interfaces are available to
  - CAMAC
  - VME (plus VME to IBM channel)
  - Fastbus (under development).

The performance of the 3081/E on the GHEISHA, LUND and WIRCHA benchmarks is given in Table 4. The WIRCHA code can take full advantage of the pipelining and the 64-bit floating-point hardware, and thus shows a large speed-up relative to the solidly 32-bit architecture of the VAX 11/780.
Table 4

Performance of 3081/E emulator

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>VAX 11/780/FPA Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHEISHA</td>
<td>4</td>
</tr>
<tr>
<td>LUND</td>
<td>4</td>
</tr>
<tr>
<td>WIRCHA</td>
<td>10</td>
</tr>
</tbody>
</table>

4.5 Loosely-Coupled Parallel Computer (Clementi Machine)

In the summer of 1984, L3 was invited to be one of the first outside users of the ‘Loosely Coupled Array of Processors’ assembled by Enrico Clementi at IBM Kingston. Clementi is primarily a theoretical chemist, and he leads an IBM supported group doing calculations of molecular structure from first principles. However good a theoretical chemist you are, such a study needs almost unlimited computing power.

Figure 8 shows a highly simplified diagram of the ‘Clementi Machine’ as it was when we visited Kingston. The ‘host’ processor is an IBM 4381 Model 3 (about 8 VAX equivalents) running VM/CMS. Attached to the host are 10 FPS 164 processors.

![Figure 8 - A Simplified Diagram of the 'Clementi Machine'](image)

Physically, two FPS 164’s are attached to each block-multiplexor channel; the maximum data transfer rate for an IBM channel is 3 megabytes per second. The FPS software required that each FPS 164 was connected to a separate ‘Virtual Machine’ running under VM. Thus to achieve parallel processing it was necessary to set up communications between a ‘Parent-VM’ and ten ‘Child-VM’s.

Apart from the benchmark of the FPS 164’s, which I have already described, our main interest was to study how effective this machine could be for HEP parallel processing. This study required the measurement of the various overheads which would limit efficiency, followed by implementation of a parallel version of one of our benchmark programs.
The overheads are shown in Table 5. First we note that the real speed of communications over the channel is quite close to the theoretical maximum. The relative slowness of communications between processes running within the 4381 might be a surprise to the uninitiated. This slowness is due to the rigid software barriers between virtual machines which were only allowed to communicate by real I/O.

New IBM software now allows virtual machines to communicate using shared memory, and FPS have also reduced many of the overheads in downloading and starting processors.

**Table 5**

<table>
<thead>
<tr>
<th>Clementi Machine communications overheads (measured 1984)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM-VM communications</td>
</tr>
<tr>
<td>VM-FPS communications</td>
</tr>
<tr>
<td>Program downloading</td>
</tr>
</tbody>
</table>

**Parallel Computing**

L3 'production' processing was simulated by running the core of the GHEISHA Monte-Carlo simulation on several AP's, under the control of a single parent running on the host.

Each AP executed the following loop:

- receive 100 kbytes of data from the parent process,
- generate an event (taking about 170 seconds),
- send 100 kbytes of data back to the parent.

There were no synchronisation points other than at the beginning and end of the job. In other words, each AP was given more work to do as soon as it had finished, without regard to how far the other AP's had progressed.

To express the effectiveness of the parallel execution we chose the expression:

\[
Fraction \ of \ parallel \ execution = \frac{\text{elapsed time (one AP)}}{N \times \text{elapsed time (N AP's)}}
\]

In this expression 'N' is the maximum number of AP's which could have been used. In this case N is equal to the number of child processes. If the AP's can multi-process, N is the lesser of the number of child processes and the number of AP's.

Figure 9a shows the elapsed time as a function of the number of AP's for a run generating 100 events. The elapsed time falls almost as 1/N. This is confirmed by the fraction of parallel execution shown in Fig. 9b, which stays close to 100% even for N=10. This is not really a surprise, given the measured overheads and the additional fact that only 10% of the host CPU power was needed to manage the 10 AP's.

Had we tried to run the LUND generator in parallel, we would have obtained rather dismal results, since an FPS can generate a LUND event in about 40 milliseconds which is comparable with the communications set-up overheads.
4.6 Tightly-Coupled Parallel Computer (ELXSI 6400)

The architecture of the ELXSI 6400, summarised in Fig. 10, is centred round the 'GIGABUS', capable of transporting between 160 and 213 megabytes per second. Physically the GIGABUS is a backplane, up to two of which may be linked together when more than 5 CPU's are required. Up to 10 CPU's may share the GIGABUS with up to 192 megabytes of memory and with I/O processors. When we benchmarked the machine in early 1985, the CPU’s were each equivalent to between 3.5 and 4.5 VAX 11/780 equivalents; the latest version is somewhat faster.

Figure 10  The Architecture of the ELXSI 6400 Computer
The ELXSI 6400 was not conceived for parallel computing. Anyone who displays the processes running on an average super-mini will find several processes queueing for the CPU. The multi-CPU ELXSI was designed to increase throughput by migrating active processes to available CPU's.

Interest in parallel computing was first generated by a customer, Sandia Laboratories. Sandia runs very time consuming calculations on a CRAY X-MP processor. To reduce the real time spent waiting for results, the Sandia programmers use the parallel processing 'intrinsics' provided in CRAY Fortran, to make use of all four processors of the CRAY for one job.

Programming parallel processing on a shared memory system can be tricky, and there is a danger that many CRAY hours might be wasted in debugging the parallel logic. Sandia resolved this problem by persuading ELXSI that they should implement a CRAY-like set of intrinsics on the 6400, which could then be a 'cheap' test bed for CRAY programs.

I will describe the intrinsics in more detail later. At this point it is sufficient to point out that they provide good support for child creation, memory sharing, and inter-process message passing.

From the operational point of view, two features of the ELXSI make it easy to use as the foundation of a 'parallel computing service' :

1. The processors are multi-processors, and can execute a large number of tasks apparently simultaneously.

2. The operating system migrates tasks automatically to balance the load on the CPU's.

**Overheads**

Data-transfer via shared memory takes no time at all. Waking up a sleeping child or parent took about 250 microseconds.

**Parallel Computing**

In contrast to the Clementi Machine, the minimal overheads of the ELXSI 6400 made it possible to test parallel computing for a wide range of applications.

The plots of elapsed time and fraction of parallel execution versus the number of processes is shown in Fig. 11 for the GHEISHA program. The implementation was exactly the same as that on the Clementi Machine; each GHEISHA event took 130 seconds on a single ELXSI CPU. Clearly, the communications overheads for GHEISHA were negligible, and any inefficiencies must be due to interference with, and inefficient management by, the operating system. Only when the number of processes exceeded the number of physical processors, was any inefficiency noticeable, and even then the parallel execution was only slightly less than perfect.

LUND events took 70 milliseconds on a single CPU, so with 10 child processes, the parent had to be woken to attend to a child every 7 milliseconds. Figure 12 shows how the system performed. The fraction of parallel execution is always over 90%, and apart from the measurement with 9 and 10 children, it is close to 100%. No care was given to give the parent any special status (although it is possible to lock important processes in one of the 16 register sets on a CPU). The efficiency drop is probably caused by the parent sharing a CPU part or all of the time with one of its children.

The range of timing, from GHEISHA to LUND, covers most of HEP event processing. GHEISHA is typical of Monte-Carlo simulation and production processing. The LUND timing is typical of 'Data Summary Tape Analysis' where up to millions of events are read sequentially and subject to simple processing.
Since perfect results rapidly become boring, we decided not to limit ourselves only to tests typical of HEP computing, and to parallelise the WIRCHA matrix inversion at the loop level. First the matrix inversion algorithm was changed to Gaussian Elimination, which is fairly easily parallelised. This initial change reduced the speed by 21%. Then a varied number of processes was used to perform the matrix
inversion, leaving the rest of the program shell unchanged and un-parallelised. In contrast to event processing, each process helping in the matrix inversion had to wait for the others when it had finished. The result of all this is shown in Fig. 13. The imperfections are clearly visible, but are mainly due to the shell of the WIRCHA program which we did not bother to parallelise.

4.7 MIMD Computer (Denelcor HEP-1)

For the benefit of all those only marginally less well informed than I am:

MIMD = ‘Multiple Instruction, Multiple Data’

HEP = ‘Heterogeneous Element Processor’

The L3 benchmarkers visited Denelcor late in 1984. Although Denelcor is now defunct, their unusual machine remains of some interest.

The HEP-1 contains up to 16 processors (PEM’s) communicating and sharing data memories over a packet-switched network. A PEM can pipeline instructions from separate processes, and 10 to 12 processes are needed to use a PEM to the full. When fully loaded, a PEM delivers about 13 VAX 11/780 equivalents.

The support for parallel processing is in the form of Fortran callable intrinsics with similar functions to those of ELXSI. Two important differences are:

- memory access synchronisation; a process can wait for another process to fill a memory location.
- Mandatory sharing of all COMMON blocks by parent and children.

The memory access synchronisation feature lends itself well to the automatic generation of parallel code by compilers. Conversely, the mandatory sharing of COMMON blocks made it very difficult to parallelise existing HEP code.
During the L3 tests, the machine made available to us suffered many hardware and software failures. As a result the number of benchmarks we could run was somewhat limited.

**Parallel Processing**

The limited results obtained with the GHEISHA benchmark are shown in Fig. 14. There was no time to re-structure GHEISHA to avoid the conflicts produced by mandatory COMMON block sharing, so all that was measured was the elapsed time to generate 100 events by running independent jobs on one PEM. It is hard to know what to expect from such a measurement; to me, the relative lack of interference between jobs running on the same PEM is remarkable. The reported CPU usage with one job running was 10%. With seven jobs it had increased to 61%.

![Graph showing elapsed time and percent parallel execution](image)

*Figure 14* Parallel Execution of GHEISHA on the Denelcor HEP-1

The existence of a Denelcor matrix inversion subroutine made it very interesting to try loop-level parallelism within WIRCHA. The Denelcor matrix inverter was 8% slower than the original code but it allowed the programmer to specify how many parallel processes would be used. Figure 15 shows how the elapsed time varied with the number of processes. The ‘percent parallel execution’ has been calculated making the assumption that one PEM is equivalent to 10 independent processors. The results show that a PEM does not become inefficient even when executing 40 processes; the deviation from perfect parallel execution is consistent with the fraction of the WIRCHA code which was not parallelised.
4.8 LEPICS, the L3 Off-line Computer

The wisdom accumulated over the long series of benchmarks and investigations, some of which have been described above, has led us to the specification for the L3 off-line computer. LEPICS means something like 'L3 Parallel Integrated Computing System'. Although we believe that the longer term future lies with tightly-coupled parallel computers, we have to have our system fully operational by the end of 1988. On this time scale it seemed that a system with a large host and attached processors was the best choice. A summary of the specifications is shown in Table 6.

Table 6
Specifications for the L3 Off-line Computer

<table>
<thead>
<tr>
<th>Installation</th>
<th>Start during 1987</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>IBM or DEC compatible</td>
</tr>
<tr>
<td></td>
<td>≥ 12 VAX 11/780 equivalents</td>
</tr>
<tr>
<td></td>
<td>≥ 10 Gigabytes of user disk space</td>
</tr>
<tr>
<td></td>
<td>≥ 8 Tape drives</td>
</tr>
<tr>
<td></td>
<td>Printers, Terminals, Networks</td>
</tr>
<tr>
<td>Attached Processors</td>
<td>≥ 30 VAX 11/780 equivalents total 3081/E (for IBM)</td>
</tr>
<tr>
<td></td>
<td>Micro VAX (of some sort) for DEC</td>
</tr>
<tr>
<td>Host-AP Communications</td>
<td>≥ 2 Megabytes/second</td>
</tr>
</tbody>
</table>
5. COMMENTS ON PRICE/PERFORMANCE

This is not an attempt to select a 'best buy' as in a report on video recorders or washing machines. The aim of this section is to try to show what you get for your money as a function of the computer architecture chosen. It is not easy to discuss price/performance in abstract terms alone, so I have chosen to centre the discussion on the systems I have reviewed. Some of these systems are no longer available, and prices (after discount) can vary by amazing factors, particularly for special customers like CERN and universities. No price I give is accurate to better than 30%.

Having absorbed these caveats, please turn to Table 7, which summarises my price/performance comparisons. I have already commented on the variability of prices. The performance figures are mainly those measured by L3, and are, of course, totally reliable. Nevertheless, the performance of machines which depart from the classical scalar architecture depends greatly on the code, and on the amount of re-writing we are prepared to do. For most machines I have expressed this dependence as a range, and I have taken the middle of the range when calculating '$ per VAX'. For the CRAY, as for other vector computers, I cannot do this because I really have no idea what range of performance we will finally get. I have therefore been a little unfair in taking the worst case of unmodified HEP code.

In comparing computer systems it is vital to take into account how complete the system is. To emphasise the importance of this I have included an entry for 'a bag of chips', specifically a Motorola 68020. This excellent processor is nothing like a complete computer system, and its price is about as relevant as the price of copper or silicon.

A 'complete' system includes disks, tape drives, printers, and terminal lines. Systems which are close to complete have been given four 'blobs'. The CRAY is normally sold with a limited peripheral configuration, expecting that most of the more trivial (but by no means cheap) I/O operations will be handled by, for example, an IBM mainframe as a front-end. This fact makes the CRAY lucky to get three blobs.

The FPS 164, which is designed to be an attached processor, but can access its own disks, gets two blobs, whereas a bare 3081/E emulator merits only one. Adding a minimal sized host to the emulator raises it to two blobs. In every case I have only estimated the hardware completeness of a system. Whether it has an operating system or not is a separate consideration.

The message which I draw from Table 7 is that there are no miracle solutions. Improvements of up to a factor of 3 in price/performance may be available by departing from the classical, easy to use, scalar mainframe solution. The task facing HEP is to choose an architecture which works well for our sort of computing. The task facing computer manufacturers is to choose architectures which work well for a wide range of applications.
<table>
<thead>
<tr>
<th>Computer</th>
<th>Price</th>
<th>HEP</th>
<th>$k per</th>
<th>How complete a system?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAY X-MP 4 CPU</td>
<td>14-17</td>
<td>100 -&gt;</td>
<td>155</td>
<td>• • •</td>
</tr>
<tr>
<td>IBM 3090-200</td>
<td>6</td>
<td>40</td>
<td>150</td>
<td>• • • •</td>
</tr>
<tr>
<td>FPS 164</td>
<td>0.25</td>
<td>3-6</td>
<td>56</td>
<td>•</td>
</tr>
<tr>
<td>Clementi Machine 1984</td>
<td>6</td>
<td>40-70</td>
<td>109</td>
<td>• • • •</td>
</tr>
<tr>
<td>3081/E</td>
<td>0.033</td>
<td>5</td>
<td>7</td>
<td>•</td>
</tr>
<tr>
<td>3081/E Farm (small host plus 6 emulators)</td>
<td>0.7</td>
<td>32</td>
<td>22</td>
<td>•</td>
</tr>
<tr>
<td>3081/E Integrated System (large host, 6 emulators (LEPICS?))</td>
<td>2</td>
<td>41</td>
<td>49</td>
<td>• • • •</td>
</tr>
<tr>
<td>ELXSI 6400 (10 CPU, 1985 model)</td>
<td>2</td>
<td>40</td>
<td>50</td>
<td>• • • •</td>
</tr>
<tr>
<td>Denelcor HEP-1</td>
<td>1.3</td>
<td>7-13</td>
<td>130</td>
<td>• • • •</td>
</tr>
<tr>
<td>Motorola 68020</td>
<td>0.0001</td>
<td>1</td>
<td>0.1</td>
<td>•</td>
</tr>
</tbody>
</table>

6. SOFTWARE

Success in a computing task is by no means assured simply by taking delivery of a large quantity of hardware. This is especially true if the hardware departs in any way from the conventional scalar processing systems which we are accustomed to use.

Before I review the software problems in more detail, let me air my prejudices. Although both vector and parallel computing systems are tricky to use efficiently, I believe that computing for experimental HEP can be implemented in a much more natural fashion on parallel computers than on vector computers.

6.1 Software for Vector Computers

HEP can use two, almost distinct, approaches:

1. Totally re-think the algorithms so that ‘von Neumann’ (one thing at a time) code becomes efficiently vectorisable. The work on the Fermilab E-711 experiment track-finder is an example of this.

The likely result is clear, intelligible code which is efficient on a specific vector architecture.
2. Parallelise the logic so that many unrelated but similar operations are performed at the same time. For example, re-organise a Monte-Carlo tracking program so that all the trivial co-ordinate transformations involved in tracking N particles through one step can be performed simultaneously.

The likely result is tricky code, difficult to write and maintain, and still tied to a specific vector architecture.

I have spent years preaching the necessity of writing clear, maintainable, portable Fortran code, so both of these approaches worry me. However, if we ignore these problems, what speed-ups are we likely to get as a result of a vector-specific software effort?

It has already been demonstrated that specific algorithms can be speeded-up by around a factor of 10. How much can complete programs be speeded-up? My own feeling (and I am not alone) is that a factor of 3 is the likely maximum.

6.2 Software for Parallel Computers

The general problem — how to apply N processors to a single task — is one of computing science’s most important challenges. I think that this will still be considered an important challenge well into the 21st century.

I will only concern myself with HEP specific solutions which are far from general. However, the foundation of my approach is parallelisation by a minimal reorganisation of the computing task. In other words, I advise stepping back and thinking about the task for a few minutes, rather than rushing in with existing automatic parallelisation tools.

Rather than make myself look ridiculous by trying to create a theory of parallel computing, I will restrict myself to examples of what L3 is doing, and intends to do, to make parallel computing work.

A High Energy Physics Batch Job

Figure 16 shows the very simple structure of most HEP batch jobs. Almost invariably such jobs involve processing events. Processing one event may take minutes (for Monte-Carlo simulation) or milliseconds (for Data Summary Tape analysis) but the job structure is the same.

Since the events are independent, they may easily be processed in parallel. Manpower, organisation and sanity dictate that we should try to manage the parallel processing within one job, rather than running many jobs at once.

Parallel processing becomes trickier when we want more than one process to access a single peripheral or a single memory location. The simplest way to solve this is to allow only one process to read and write events, and fill histograms. In most cases this works perfectly, since the 'process event' kernel still dominates the CPU requirements.

On tightly-coupled machines, it is usually possible to let the parallel 'child' processes fill the histograms and statistics in shared memory. Some machines even offer 'fetch and op' instructions which require no memory locking. On loosely coupled machines it is usually better to let the host handle the histograms.
Parallel Processing Examples — Emulator Farm

L3 is now one of the users of an emulator farm comprising an IBM 4361 host, and five 3081/E emulators. The L3 Monte-Carlo event simulator has been parallelised in a rather simple way so that events can be generated on an arbitrary number of emulators, but program initialisation and termination, and all I/O, are done by a single process on the host.

The host currently accesses the emulators through CAMAC. This works perfectly for Monte-Carlo simulation. For event processing with a higher I/O to CPU ratio, and with more work to be done by the host, a more powerful host and faster host-emulator communications will be needed.

From this experience, we know the software tools needed to make such a system work. We also have a fairly good idea about the additional software facilities needed to make a host-AP system into an integrated HEP computing environment.

Here is a commented list of the requirements:

1. Cross compiler. (exists)

This is provided by the combination of the IBM compiler and the 'translator'. (Note that the 370/E emulator, developed by the Weizmann Institute, Rutherford Lab. etc., operates directly or IBM object code).
2. Cross linker. (exists)

3. Debugging Tools. (some exist)

The best place to debug the Fortran code is on the host system. It is also advisable to debug the parallelised version, using a system like VM-EPEX (VM Environment for Parallel Execution), to simulate a multi-processor system without using real emulators.

After debugging on the host system, the code should run on the emulators without problems. This will probably be almost true, eventually, but at present occasional problems with the translator or with the emulator hardware are tricky to debug and are best handed over to the few experts.

4. FORTRAN calls to: (exist)

- download a program module
- download data
- start emulator
- wait for emulator to finish
- upload data
- interrupt emulator

5. Resource Management (does not yet exist)

- assign emulators to requesting jobs on the basis of the job's relative priority
- request a job to release (some) emulators
- forcibly remove emulators from uncooperative jobs

It makes an enormous amount of sense to put emulators on an IBM compatible host and avoid any data-format incompatibilities. With some types of attached processors, for example a 68020 on a card, format incompatibilities are nearly inevitable. This should not be a problem for well organised software. For example, all L3 software uses the ZEBRA data structure manager. At any stage in the processing, ZEBRA data can be transported to another processor in a machine-independent format.

Parallel Processing Examples — Tightly-Coupled Systems

Tightly-coupled systems can do everything an emulator farm can, but the lower overheads make it attractive to try to parallelise a wide range of tasks. When the 'process event' kernel shrinks to a few tens of milliseconds, the programmer has to be careful even on a shared memory machine.

Discipline in the use of shared memory must usually be imposed by hand; one process must tell another that the shared memory is available. To squeeze out the last 10% of performance, the programmer may have to make his own decision about which data may be held in a processor's cache memory, and which data must always be read from and written to the main shared memory.

'Fetch and op' instructions (e.g. fetch and add) can be used to implement inter-process synchronisation, and for common histogram and statistics accumulation without explicit discipline.

An emulator farm operating system has to take great care to ensure that the number of processes exactly matches the number of processors. Most tightly-coupled processors can multi-process, so the operating system can exert a much looser control over the total number of processes.

To make some of these points a little clearer, I list below the Fortran 'intrinsics' available to support parallel processing on the ELXSI system. Figure 17 shows how these intrinsics can be used to set up a simple parent-child system, which may be readily generalised to an arbitrary number of children.
Figure 17  A Parent-child System Using ELXSI Intrinsic

ELXSI Parallel Processing Intrinsic

Most applications need only the following calls:

STATUS = MT$ShareMemory (Address, Length, Cacheable?)

MT$SetupSemaphore (Name_of_Semaphore, QueueLength)

MT$SetupTasks (Number_of_Tasks)
creates identical copies of the parent at this instant.

MT$StartTask (Task_no., Subroutine_Name, Subroutine_Arguments)
calls a subroutine within a child process.

MT$SignalSemaphore (Semaphore_Name)
icrement the named counting semaphore.

MT$WaitOnSemaphore (Semaphore_Name)
if the named semaphore is greater than zero, continue, otherwise sleep until
the semaphore reaches zero. (Sleeping processes consume no CPU cycles.)
MT$WaitOnTask (Task_no.)
sleep until the child task RETURN's (or STOP's) from the subroutine
called by MT$StartTask.

MT$DestroyTasks ()
kill the children.

MT$WhatTaskAml ()
returns its task number to a child.

Enthusiasts can also play with:

MT$SetupLock (Lock_Name)

MT$Lock (Lock_Name)
if the named lock is open, lock it and continue,
if the named lock is locked, spin in a NOP loop
until the lock is opened, then lock it and continue.

MT$Unlock (Lock_Name)
open the named lock.

MT$FlushMemory (Address, Length)
flush the cache-memory belonging to the calling process to ensure
that the main memory contents are updated.

MT$FlushMemoryAllSharers (Address, Length)
as above, but for all processes sharing this memory.

Parallel Processing Invisible to (Most HEP) Users

After reading the previous section, and struggling to understand what the meaning of locks and
semaphores might possible be, an HEP user might be forgiven for abandoning parallel processing in
total disgust. In reality, most users (even those who write analysis programs) need hardly notice parallel
processing at all.

I will describe briefly the simple program organisation which makes this possible. More ambitious
and more flexible schemes are under discussion but I have to keep these lectures to a finite length.

The keys to invisible parallel processing are clearly visible in the example of an ELXSI implementa-
tion shown in Fig. 17. Although the parallel processing logic can be moderately complex, it does not
of the programming effort goes into writing code which fits cleanly within one of these phases, and the
writer of this code can forget about the parallel processing environment.

The overall parallel processing framework can be generalised and made portable by inventing a
set of generic parallel processing intrinsics which collapse to nothing on a scalar machine, and call the
appropriate machine dependent services elsewhere.

All this sounds too good to be true, and to some extent it is. Adapting many existing programs to
such a scheme would be a nightmare combination of major surgery and nasty little fixes. L3 is in the
fortunate position of having few existing programs, and we can ensure that further software development
obeys principles which will make it suitable for parallel processing. The principles are quite simple:

- it must be easy to identify the data-flow into and out of a processing ‘module’. Ideally the input
  and output should each be a single data-structure.
any writes into variables which are not private to the event being processed must be clearly identified and localised. Filling of common histograms and statistics is the most obvious example of this.

7. CONCLUSIONS AND SUMMARY

I have shown that, in spite of the continuing fall in the cost of MIPS, HEP computing needs are growing so rapidly that they need either more and more money, or a more imaginative approach. Furthermore, not even money can save large CPU users like HEP from parallel processing in the near future.

In some sense, all the more imaginative approaches involve computing in parallel. In vector and deep-pipeline computers the parallelism is on a microscopic scale, but these machines do not appear to be well matched to HEP needs. Using the larger scale parallelism of tightly- and loosely-coupled parallel systems normally requires intervention by the programmer, but these architectures match well the intrinsic parallelism of HEP computing.

For immediate relief for under-financed HEP experiments, loosely coupled host-AP systems appear a natural choice. The longer term future of commercially successful parallel computing probably lies with much more tightly coupled, and thus much more versatile, machines.

Parallel computing could be viewed as the solution of the hardware problems by the creation of software problems. In the general case, this is not so inaccurate, but in HEP computing it is quite possible for most programmers to ignore the parallel processing framework.

REFERENCES

WHAT IS THE ‘OBJECT’ IN OBJECT-ORIENTED PROGRAMMING?

O.M. Nierstrasz

Centre Universitaire d’Informatique, University of Geneva, Geneva, Switzerland

Abstract

Object-oriented programming has become quite widespread in recent years, although there are few guidelines to help us distinguish when a system is “truly” object-oriented or not. In this paper we discuss what have emerged as the main concepts in the object-oriented approach, and we attempt to motivate these concepts in terms of how they aid in software development.

1 Introduction

There is much confusion about the term “object-oriented”, which it appears can be applied to anything from an operating system to an interface for a text editor. Although there are a number of things the term normally brings to mind, there is no clear way of telling whether system A is “really” object-oriented, and system B is not.

In this paper we will discuss a number of object-oriented “concepts”, and put forward the hypothesis that object-orientation is an approach rather than a specific set of language constructs. In fact, one can probably use an arbitrary programming language and still write in an object-oriented style. However, object-oriented programming languages do exist with built-in constructs that support (or enforce) this style of programming.

The principle that is fundamental to the object-oriented approach is encapsulation, or more precisely, data abstraction: an “object” packages an entity (data) and the operations that apply to it. In its most basic form we may have a module of a program consisting of a number of static variables together with the set of procedures that are used to manipulate the variables. In general, those variables are “private”: the object is defined by its behaviour, not its representation. So if I have an object called “Jaguar”, I do not change its colour by modifying its “colour” variable, but by executing an operation such as “paint”.

The notion of data abstraction is especially useful if one can apply it to multiple instances of an object “type”. A language like Fortran provides a fixed set of data types, such as integers and real numbers, each of which supports a number of operations (addition, subtraction ...). Languages like C and Pascal support the construction of programmer-defined types, one cannot define new type-specific operations. Object-oriented languages enable the programmer to create his (or her) own object types. Each object is then an instance of some type, and supports the operations (called methods) that are defined for that type. (The word “class” is often used interchangeably with the word “type” when talking about objects, however, it is more precise to think of an object class as the set of instances of an object type.)

Object “independence” is the notion that each object is ultimately responsible for itself. Objects can only be manipulated via their methods.
Supporting this idea of object independence is a model for object communication which replaces the notion of operating on objects with that of passing messages to them. Every method invocation is seen as a message requesting the object to perform some action. The method is (conceptually) not performed by the caller, but by the object receiving the message. It follows, then, that the recipient of such a message is, in some sense, free to refuse the request, or choose from a variety of possible implementations of the method.

Classes of objects may not be disjoint. Object types may form a hierarchy in which a subtype "inherits" the methods of one or more parent types. The implementation of the subtype's methods may or may not be inherited from the parent. In fact, one may even have multiple implementations of a single type which only share their interface. By properly organizing such a type hierarchy, one may concentrate on the similarities between objects rather than on their differences.

Fully object-oriented systems have a strong degree of homogeneity. This means that everything is an object. One does not distinguish between, say, programs and processes and files and objects – one only has objects, and objects are both active (like processes) and persistent (like files). In particular, the object types (or, more precisely, the type specifications) are objects of the type called "type".

A natural step to take is to map objects to an "actor" model of computation. Simply put, an actor is a cheap, persistent, message-passing process. (The actor formalism as described in [11] is more rigorous, however this simple definition suffices for our purposes.) An actor must be inexpensive because we would like to be able to easily create and destroy them. If there is a one-to-one mapping between actors and objects, then it is clear that we may have many more actors than there are processes in a traditional multiprocessing environment, though at any one time only a few actors will actually be doing anything. The message-passing paradigm supports the notion of actors communicating with one another whether they are running on the same machine or not. If we consider objects to be active entities running "on top of" actors, then we see that the object-oriented approach also provides us with an easy way to understand concurrency, and applications running on a network of machines. At the same time, we note that there are several object-oriented languages and environments that do not support the notion of concurrently executing objects.

The main object-oriented concepts, then, are:

1. Data abstraction.
2. Independence.
4. Inheritance.
5. Homogeneity.
6. Concurrency (some systems).

We shall elaborate on each of these concepts in turn, and attempt to explain their importance.
2 Data abstraction.

By far the most important concept in the object-oriented approach is data abstraction. By this we mean that we are interested in the behaviour of an object rather than in its representation. Every object has a clearly defined interface which is independent of the object's internal representation. The interface is a collection of operations, or "methods", which may be invoked by another object.

Furthermore, one may have many instances of an object type. The programmer is not restricted to a fixed set of types, but is free to add new types as required. This capability first appeared in the programming language "Simula" [3], a simulation language in which the programmer was able to define his own "classes" of objects (i.e., corresponding to the real objects one wished to model).

A type definition looks very much like a module: there are a collection of permanent variables that encode the "state" of the object, and a set of methods that use and change that state. A module, however, only defines a single instance, whereas one may have many instances of a new type. In order to make use of an instance of that type, one only needs to know what the interface is, namely, the names of the methods, and the types of the input and output parameters. (Naturally some specification of the semantics of the methods is also required.)

The most immediate benefit of this approach is that the programmer is free to use higher levels of abstraction as appropriate. (I.e., at each level of abstraction one concentrates on that level's functionality, while hiding the lower-level details of implementation.) One is encouraged to decompose a programming problem into a collection of cooperating objects of varying levels of complexity.

The separation of interface and implementation of a new type means that types are, to some extent, representation-independent. This makes it easier to experiment with different implementations of an object type, and increases the portability of software. It is, however, crucial to design an interface very carefully, since that is the part of an object type that remains stable across different implementations.

A language that allows programmers to define their own types is a natural candidate for strong typing. Type-checking is done as soon as possible, usually at compile-time. (Sometimes one does not know at compile-time what kinds of objects will be manipulated, so one must do explicit type-checking at run-time; for example, a mailbox object may hold various types of mail objects.)

In an object-oriented language, one is typically allowed to use the same method names for a variety of different object types. This is called "overloading", or polymorphism. For example, one may wish to define addition for complex numbers, and use the symbol "+" as the method name, even though it is already defined for integers. The type of the object to which such a method applies is, in a sense, an implicit argument to the method. Without overloading, one would be forced to define the methods "complex_add", "integer_add", and so on. Overloading allows for more readable code. If used in an undisciplined fashion, however, overloading can have quite the opposite effect. (As a grotesque example, consider the use of the symbol "+" as the method name for complex subtraction.)
Well-designed objects are typically applicable in a number of situations. One can therefore expect in an object-oriented environment that most of the object types one defines can be re-used, or, conversely, that when putting together an application, one can find many of the parts one needs already available as pre-defined object types.

3 Independence.

There are several notions of independence displayed by objects. The most fundamental is that objects have control over their own state. An object’s methods are the only interface to its state. Since this is not always guaranteed in languages that enable programmers to add new data types, it is worth emphasizing. Clearly, if one has access to the internal representation of an object, then one loses the property that the implementation is independent of the interface.

Objects also have control over their own existence. Once created, an object will continue to exist (or persist) even if its creator dies. Persistent objects eliminate the need for “files”. One need not be concerned as a programmer with the location of an object in main memory or secondary memory. Ideally, dormant objects automatically “migrate” to passive memory. Garbage collection, however, is compatible with this notion, since an object that is neither active nor accessible is effectively dead already.

Another form of independence is the ability to add new object types at run-time. This is crucial if the object-oriented environment is also the development environment. Otherwise the addition of new types must take place “outside”. If new types are to be added dynamically, then old objects must be capable to talking to instances of the newly-created types. (Compile-time type-checking is clearly not possible for those objects that will talk to objects of types that do not exist yet.)

4 Message-passing paradigm.

Independence of objects is supported conceptually by using message-passing as a model for object communication. In terms of the model it is clear that an object cannot “operate on” another object. The only way an object can interact with the outside world is by sending messages. Consequently, object A “invokes” a method of object B by sending B the message, “please execute this method”.

Assuming all goes well and the message can actually be delivered, B is free to interpret the message as it pleases. It may choose amongst several ways of handling the request, it may delay responding, or it may decide that it does not wish to handle the request at all, and return an exception report. “Returning” is also accomplished by message-passing.

It is important to realize that message-passing is a model for object communication rather than an implementation requirement. In fact, more than one object-oriented language translates message-passing into straightforward procedure calls. This is possible when flow-of-control is single-thread, and no more than one object is “executing” at any time. However, in a concurrent environment where objects are truly active,
"real" message-passing is a natural way in which to implement communication between objects.

Message-passing may be synchronous or asynchronous. Basically, the difference is that with asynchronous message-passing the message is put on a queue, and the sender is immediately free to concentrate on another task (though it usually just waits for a response). With synchronous message-passing, the sender blocks until the message can be delivered, sometimes even until the recipient prepares a response. The problem with asynchronous message-passing is that it is (strictly speaking) impossible to implement, since there is no bound on the size of the message queues. The problem with synchronous message-passing is that it says nothing about the order of arrival of messages when multiple senders compete for a single recipient, and it limits the amount of concurrency by blocking the sender.

Asynchronous message-passing is perhaps more consistent with the idea of object independence, but it is not a pre-requisite for an object-oriented system.

5 Inheritance.

From the point of view of organizing software and reusability of code, the most important mechanism in an object-oriented language is the ability to specialize object types. A specialized type (a subtype) inherits the properties of its parent type, then possibly adds more properties.

A program source, for example, may be seen as a specialization, (or refinement) of a text object. An instance of a subtype responds to all the messages that the parent type understands. The reverse is not true. A subtype inherits the method interfaces of the parent type, but may add new methods that the parent cannot respond to. In addition, a subtype usually inherits the implementation of the parent by default, that is the set of instance variables that encode the state of a parent instance, and the implementations of all of the methods. The subtype could add new instance variables and methods.

A subtype may, however, have a different set of instance variables, and quite different implementations of the inherited methods. For example, real numbers are a refinement of integers, but they are represented and implemented quite differently in a computer.

The set of all object types forms a specialization hierarchy with the type object at the root. With simple inheritance, this hierarchy is a tree.

If multiple inheritance is supported, the hierarchy is, in general, a directed acyclic graph. This means that a subtype may have more than one parent type (but no type can be its own ancestor!). The parent types in this case are called "traits" [7] or "flavours" or "mixins".

The difference between a trait and a type (if any) is that a trait specifies an interface, but it need not define an implementation. One reason for this is that a trait may not be of interest as a type on its own. Consider for example the trait "ownership". Such a trait might specify methods for setting the owner of the object, changing ownership, and determining access rights. Ownership is only of interest in conjunction with other properties. An "owned object" would be of little use on its own.
Multiple inheritance increases the reusability of software. Simple inheritance is inadequate for describing many of the useful relationships between object types. Suppose that we would like to have both owned and unowned pieces of text. Similarly we might have owned and unowned graphical objects. (An unowned object might be part of a larger owned object.) The type owned_text is a refinement of text, and owned_graphic is a refinement of graphic, but with simple inheritance there is no way to say that the methods pertaining to ownership are to be shared. Multiple inheritance encourages the programmer to define traits that can be combined much more flexibly.

The added flexibility is not without its cost — there are no easy solutions if one wishes to combine two traits that use the same method name to mean different things. Provided there are no naming conflicts, multiple inheritance is an elegant mechanism for aiding in the design of new object types.

Variation of types is another form of inheritance. A new type may be defined as being equivalent to an existing type, adding no new methods. The type definition serves mainly to ensure correct use of objects of that type — type clashes can be detected by the compiler, or at run-time. One would not wish to confuse names of people with names of programs, even though they would both be implemented as strings.

Furthermore, equivalent types may have implementations tailored to a particular environment or usage. Window objects may behave the same on a variety of graphic devices, yet will be implemented in different ways. Similarly one might choose from a variety of implementations of lookup tables with different overhead and performance characteristics, though all are functionally equivalent.

A weak form of “inheritance” is parameterization. A “container” object is an object that manages objects of a given type (or set of types), yet is not concerned with what that type is. When an instance of a container is defined (or created), the type of the contained object should be known. In the definition of the container type, the type of the contained objects is specified as a parameter. For example, one would define an array as an array of a contained type. When declaring an array, one would specify an array of integers, or an array of mail messages. (The array object does not care what it manages, but at some point the compiler needs to know what it will be.)

6 Homogeneity.

In a “fully” object-oriented environment, “everything” is an object. One does not distinguish between programs and objects — objects are the active entities. Object types (or rather the specifications and implementations of object types) are objects too (of type type). This is necessary in a complete environment, since a programmer will need to instantiate new types from within the environment.

To say that “everything” is an object is formally attractive, but one must take care. For example, are messages objects? If so, in order to manipulate a message, is it necessary to send it a message? This sort of circularity is normally broken at some level. It is fundamental to the object-oriented model that messages can be sent between objects, without having to define “how” this is done.

More important is the ability to think of instance variables as objects. This enables
a programmer to construct complex objects whose parts are also objects. Again, the circularity must be broken at some point. Is it really useful to talk about the “parts” of an integer as objects? What are the instance variables of a single bit? In any case, it is the behaviour of objects that is crucial, not their representation. We should, therefore, not feel uncomfortable with the idea that certain basic object types are given, and all other types are built up from them. There is no need for us to know how the basic types are implemented.

The principle of homogeneity can be carried even to the level of code. In Smalltalk, control structures are implemented using objects. A block expression is an object that can be executed by sending it the message value. Conditional execution is accomplished by sending an object of type boolean two block expressions, one to be evaluated if the value of the boolean is true, the other if it is false. Similarly, iteration is accomplished by sending an integer the message timesRepeat, with a block expression to be executed as an argument.

Carrying the notion of homogeneity to this degree certainly makes for a very consistent view of the environment, and it means that interpreted (rather than compiled) code can be understood in terms of communicating objects, but it also means that programmers may have to change the way they think about problem-solving at even very low levels. Does it help a programmer to think of the expression:

\[ 1 + 2 \]

as “send the message ‘+ 2’ to the object 1”? Probably not. At the same time it is reassuring that the object-oriented approach is sufficiently powerful to capture computation at whatever level of abstraction.

7 Concurrency.

The message-passing paradigm of object communication lends itself well to an environment in which every object is an active entity, i.e., a process. Since there are large numbers of objects, one must be able to have large numbers of processes (most of which are normally “asleep”). Furthermore, since objects are persistent, it is desirable that the processes that implement them be persistent too.

We call such processes actors, after the computational model invented by Hewitt [11]. The actor formalism can be useful for understanding object interactions. From a purely practical point of view it is attractive to think of objects being implemented “on top of” actors. Actors, then, are inexpensive, persistent, message-passing processes. Objects add structure to the state of an actor, and provide structure to the message-passing that may take place. (For examples of actors as they appear in a programming language or an operating system, see [4] or [15].)

Message-passing can be defined in several ways. We have already discussed synchronous and asynchronous message-passing. In addition, an actor may be able to receive messages at a number of different ports. Ports typically have different priorities, so that messages sent to a port of high priority will be received before those waiting
at another port. By analogy, people can receive messages by letter or by telephone. Telephone messages have higher priority than letters, though a second telephone call will not interrupt a call in progress.

Multiple ports are not strictly required, though they do provide us with a mechanism for "interrupting" an actor while it is executing.

Even with single-port actors we can model many kinds of object interactions. Object A invoking method M of object B can be easily understood as follows: Object A sends a CALL message to object B. When B receives the message, it prepares a response and sends a RETURN message to A. If there is a problem with A's call, B sends back an EXCEPTION message instead. Furthermore, if A itself receives other CALL messages before B's response, those messages are delayed until B returns.

B's response could also be directed to a separate port of A, thus avoiding the need to delay messages.

B might be acting as an agent for another object C. In that case, B would forward A's message to C (including the information C needs to send the response). C could then send its response directly to A, rather than through B.

These protocols may be modified to allow, for example, recursion. Suppose that B wants to call some method of A before sending its response. If A delays all calls because it is expecting a response from B, then B's call will never return. Recursion could be permitted by having A accept calls that it recognizes as originating from its own call to B. A "token" is thus passed around, which A will recognize when it receives a CALL message from B, or another intermediary.

We can model transactions with message-passing as well. A transaction is a sequence of operations on a collection of objects which is performed atomically, that is, either all of the operations are performed, or none of them (in case the sequence is aborted), and the intermediate states are not visible to other objects. Object A would, for example, ask object B to ignore calls from other objects while the transaction is taking place. A would then be guaranteed that B is not seen in an inconsistent state. By analogy, one can imagine the objects involved in a transaction as having a meeting behind closed doors. Anyone who is requested to join the meeting may enter the room, but no one may leave the room until the meeting is adjourned.

Typically one may also "back out" of a transaction – if the meeting is aborted, then the participants return to their state before the meeting started. The CALL/RETURN message-passing protocol is followed as before, except that after returning, each object enters a "ready" state in which it is prepared to either commit or abort. At any time an object taking part in the transaction may issue an ABORT, which causes all participants to back up to a checkpointed state. (This message is broadcast along the already-established CALL/RETURN paths.) If the transaction completes without an abort being issued, the initiator of the transaction commits it by sending a COMMIT message to the participants (again, along the paths that the CALL/RETURN messages took).

Nested transactions are a useful concept (meetings within meetings). Each object would then keep a stack of checkpointed states, one for each nesting level. (Note that it is not necessary to completely save each state, but only those parts that may be
modified in the transaction.) Argus [12] is an example of a programming language that supports both data abstraction and nested transactions.

Deadlock is still a real possibility. Consider two independent meetings, each of which suddenly requires the participation of someone in the other meeting.

Also note that the success of the two-phase commit protocol outlined above assumes reliable message-passing and checkpointing. A real implementation would have to address these issues in an object-oriented environment, as it would in any other setting.

As a final example, we can model triggering of activities by message-passing. Object A may send a SET trigger message to B, that says, "let me know when x happens". Later, when x happens as a result of yet another object making a request of B, B may send a NOTIFY message to A. When A receives the notification it initiates a new activity. A may also UNSET the trigger, telling B that it is no longer interested in x. A general triggering mechanism like this enables applications to know about interesting events without having to poll for them.

The examples given in this section were not intended so much to show what kinds of mechanisms are required for a concurrent object-oriented language or environment, but rather to indicate what the possibilities are, and how easily independent, message-passing objects can be mapped to an actor world. Since message-passing is the only medium for communication, it is also a natural step to map actors, and therefore objects, onto a heterogeneous environment of different processors connected through a network. See [8] for a discussion of similar issues in an environment that supports synchronous message-passing.

8 Concluding remarks.

The fundamental concepts in an object-oriented environment are data abstraction and inheritance. A programmer specifies an interface to a new object type as a collection of methods which can be invoked. Separately, one defines the representation of the object's state, and the implementation of the methods. The methods should be the only acceptable interface to an object.

One must be able to create multiple instances of an object type.

A subtype inherits the methods of its parent type, but not necessarily the implementation of the methods. If multiple inheritance is supported, a subtype may inherit methods from several parent types.

The inheritance mechanism "overloads" methods, since methods apply to objects of different (though usually related) types.

In a fully object-oriented system, types are also objects, and one can instantiate new types at run-time. In a homogeneous design, expressions in the programming language are also objects, as are instance variables of an object's permanent state, and even the messages objects use to communicate with.

If concurrency and distribution are at issue, each object may be an active entity that sits "on top of" a message-passing process, called an "actor".
Examples of fully-integrated languages are Smalltalk [1, 9, 10] and Loops [13]. Smalltalk does not support concurrency, distribution or multiple inheritance, but it does provide a strongly homogeneous view of its language, in which even control structures are objects.

Languages like Ada [2] support data abstraction, but no multiple instantiation or inheritance. It is impossible to add new types at run-time.

Some languages like Lisp and C have been enhanced to give them some object-oriented capabilities, without taking away anything that was in the language before. In the case of Lisp [14], "flavours", or traits have been added, so that one has both data abstraction and multiple inheritance. With Objective-C, (or its predecessor, OOPC), [5, 6] a pre-processor was written that translates Objective-C object classes into C code, that can then be compiled by a regular C compiler.

These two examples suggest that it is possible to program in an object-oriented "style" without actually using an object-oriented language. In fact, this can be done, though one would then not be able to make full use of inheritance or overloading of operations. Nor would one be able to add types at run-time unless an interpretable language (like Lisp) were used.

As with any other programming style (procedural, applicative, logic-oriented ...), one requires a programming discipline, or methodology. With object-oriented programming, perhaps the most difficult task is deciding how to naturally decompose a problem into objects. How does one design object types (i.e., interfaces) for maximum reusability and applicability?

In summary, the issues addressed by the object-oriented approach are:

1. Maintainability, through the separation of interface and implementation.
2. Reusability, as a function of well-designed general-purpose object types, and through the mechanism of inheritance.
3. Reliability, through strong type-checking
4. Portability, again, through the separation of interface and implementation.
5. Concurrency and distribution, through object independence, and the message-passing communication model.

References


OBJECTS IN ADA AND ITS ENVIRONMENT

I.C. Pyle

Systems Designers Scientific, Farnborough, United Kingdom

This course of two lectures for the 1986 CERN Summer School focusses on two distinct kinds of object of importance in programming: the data objects within the program (that model real objects in the external world), and the units of program text that are manipulated in the design and maintenance process. The Ada* programming language sheds light on both of these, as well as clarifying a number of misconceptions about programming. The first lecture deals with some important features of Ada itself, concerning data objects in programs. The second lecture deals with Ada program development, concerning the program objects that are handled in the Ada Programming Support Environment (APSE).

1. Important features of Ada

Ada was invented to deal with the problems of programming embedded-computer systems, that is, those in which a computer is embedded in some larger system, automatically sensing conditions and controlling effectors. This is a more demanding situation than conventional general-purpose programming, and Ada has become progressively more widely used as its power has been appreciated. It has captured the 1980 state of the art in Software Engineering, and provides the concepts and notations for most of the needs of programming. As well as real-time techniques, for handling the concurrency, application-specific input/output and fault tolerance of the original application area, Ada supports modern software engineering methods, which are needed to improve program correctness and visibility. Ada is now recognised as the major programming language of the decade, which is suitable for programming all kinds of information-handling systems.

1.1 Data Objects

An embedded-computer system consists of a computer interacting with some part of the real world, through sensors by which it obtains information and effectors by which it causes change to happen in the real world. Of fundamental importance, therefore, are the objects in the real world about which the embedded computer is concerned: what information is obtainable about them by the sensors, and how they can be influenced by the effectors. Understanding the real objects, their interactions and their relationships with the computer, are fundamental to system design.

Precisely the same principles apply to any real computer system, even one that is not strictly 'embedded'. Any useful computer system is concerned with some aspect of the real world, and needs to acquire information about

* Ada is a trademark of the U.S. Government (Ada Joint Program Office).
that aspect in order to produce some information that will be used for the purpose of the system. Again, the objects in the real world are of fundamental importance.

We record our understanding about real-world objects in the form of models, that capture the facts we consider relevant about them. Some of these facts are structural and permanent (such as the fact that a ship has two position coordinates and two velocity coordinates), while others are ephemeral or dynamic (such as the fact that a particular ship is currently at position 30°W 0°N heading 270° at 15 knots). We build programs to handle data objects, as models of the real-world objects.

Ada (in common with most other modern programming languages) makes an absolute distinction between the permanent and the ephemeral properties of a data object. The permanent structural properties, which determine what kinds of dynamic properties it might have, are called its type, and are entirely fixed by the program. Recognising the importance of data types has been perhaps the most important advance in programming technology in the 1970s. The type of an object expresses not only the set of values it may have (with particular precision) but also the interactions and communication that may take place between objects of that type and other objects of the same or different types. It is also important for considerations of performance and efficiency in programs: the representation chosen for each type strongly influences the space and time needed for the model to operate. Ada has strong typing, which means that each data object has a unique type and all types are disjoint. This provides powerful checking of consistency at compile-time.

Earlier high-level languages had fixed sets of types (usually integer, floating point, Boolean and perhaps character), determined by the built-in operations of the computer. Programs in these languages are written using variables of the built-in types (almost always scalar quantities) and loosely bound collections of variables for associated quantities. Ada gives much greater prominence to data types, so that the programmers’s attention is focussed on a higher level of abstraction, after which specific variables and constants of the application-related type can be introduced.

1.2 Data Types

Ada uses two principles for making data types: structure and constraints. Structure deals with multiplicity of components in a type, whereas constraints deal with limits on the variety of values for a single component. Structures may be recursive, so that what is a single component in one data type may itself contain inner components. Structures may be static or dynamic. Constraints apply only to particular kinds of data type; they may be combined, with tighter constraints imposing on looser ones. Scalar values need not be numerical: they may also be enumerated identifiers or characters.

The most important structure is a record: a set of labelled components each of which has a particular type. The values in the type are the Cartesian product of the values in the component types.

```plaintext
type SHIP is
record
  LAT: DEGREES;
  LONG: DEGREES;
  HEADING: DEGREES;
  SPEED: KNOTS;
end record;
```
There are also *variant records*, with some components present only in certain circumstances, and *access types* that provide dynamic structures (like pointers in other languages). Ada has arrays, of arbitrary dimensionality, in which any type with discrete scalar values can be an index, and the elements of the array can be of any type at all.

The most important constraint is a *range*, applied to a scalar type such as an integer, floating point or enumeration type. By means of this technique, Ada cuts the Gordian knot of program portability, where a choice must be made between efficiency on a particular computer and the flexibility of machine independence. Mathematical integers are unlimited in range; computer implementations have ranges for single-length, double-length or multiple-length integer working; but particular problems need integers (and other scalar values) within specific bounds. Ada gives the programmer the choice: either to introduce a problem-specific type with an appropriate constraint, or to use a computer-specific type with its implied range. The former is normally preferable, and gives machine independence, since the compiler chooses whether to use single length or double length working on the target computer, in order to obtain the required range. The latter method, however, is important in some circumstances, for example when we do not want the expense of double length working, but would prefer the program to be rejected by the compiler if the target word length is too short.

Similar ideas underlie other kinds of constraint: the precision of floating-point and fixed point numbers. The programmer can state the precision the problem requires, and the compiler will provide the best implementation of that precision (or better) available on the target computer.

```plaintext
type KNOTS is digits 2 range 0.0 .. 60.0;
```

Fixed point types must always have a constraint on their granularity (known as *delta*) and their range. Further contraints are available for discriminating among variant records and for giving the index range for an array.

In all of the treatment of types in Ada, it is recognised that there may be structurally similar but logically distinct types. Within a particular type, there may be subtypes (that can have more limiting constraints than the base type) which may be used in the same way as their base type, but with additional checking.

1.3 *Exceptions*

The possibility that constraints might be infringed during execution of a program is one of the psychological break-throughs in Ada. In previous languages, perfect execution was presumed, and if (surprise, surprise!) something went wrong - usually an array bound excursion - the program ran amok, leading eventually to a hexadecimal dump. Ada, in contrast, presumes that any action might fail. This could be because of a constraint infringement, logical inconsistency, or for any other reason determined by the programmer. The exception mechanism allows the programmer to deal with the situation.

The problem of failure in a program breaks down into three parts: how to detect that the exceptional situation has arisen; what immediate action to carry out in order to recover; and where to resume normal operation after the recovery. Ada's exceptions deal with all three.
An exception is a situation that is not intended in normal operation, but is recognised as a possibility that might happen. Defensive, or fault-tolerant, programming deals with these situations. In real embedded-computer systems, as much effort can be spent on the defensive programming as on the normal operational parts.

There can be many kinds of exception. Some are built in, such as CONSTRAINT_ERROR and NUMERIC_ERROR, but most will be specific to the problem domain:

    exception COLLISION_COURSE;

For each exception, the program can contain one or more exception handlers, that are executed when the exception arises. (The choice of which depends on where they are placed in the program, which we will not discuss in detail.) The ordinary program may detect an exception explicitly

    raise COLLISION_COURSE;

or implicitly

    CC := CC + 1; -- with initial CC = 80

whereupon that part of the normal program is abandoned (incomplete), and the handler executed instead. Since this diversion from the normal execution sequence is considered to be a rare event, the overhead of linking to the appropriate handler is tolerated (in order to avoid overhead in the normal situation when no exception arises). The handler is written just like an ordinary piece of program, and allows the programmer to prescribe whatever recovery is appropriate: try the action again, try an alternative action, ask for operator assistance (if there is an operator), put a dummy value in, or close down the plant etc.

    when CONSTRAINT_ERROR =>
        CC := 0;

When all of the handler has been executed, the program continues as though the part containing the handler had been completed normally. Thus the programmer controls, by the way the program is structured, the overall shape of execution of the model.

1.4 Program Structure

Ada has three kinds of program unit: subprograms, tasks and packages. Subprograms are essentially the same as procedures, subroutines or functions in other languages. The only differences concern the use of parameters (subroutine dummy arguments): in Ada, each parameter has a mode which may be in, out or in out, determining the direction the parameter value is passed between the caller and the subprogram. Parameters of mode in can have default values, which means that the caller may omit them. Subprograms are recursive and reentrant.

Tasks introduce a major innovation in programming languages (foreshadowed in Algol 68 but omitted from Pascal), recognising that programs need not be sequential or even deterministic. Each task in an Ada program prescribes a single sequence of actions for execution, but distinct tasks proceed independently of one another, except when they explicitly interact. The fundamental interaction between tasks is called a rendezvous, being the common action of two tasks at which data may be exchanged. From this
fundamental interaction, any desired kind of communication between tasks may be constructed - message-passing, transactions, readers/writers etc., as appropriate to the specific problem and chosen design. (Note that Ada does not use semaphores or signals, which have been found to be error-prone, nor fork and join, which work on a different principle of tasks not necessarily being single sequences.)

As well as serving to describe parallel programming, tasks also include timing (the delay statement and time-outs) and non-determinism (where the actions taken by the computer depend on the relative timing of distinct tasks). Also it is important to appreciate that tasks ultimately control strictly sequential actions, where reentrance must be prevented. Tasks, with appropriate ways of accessing input/output devices, are also intended to be used as interrupt handlers and device drivers.

The third kind of program unit in Ada, and perhaps the most important, is the package. As we shall see in the second lecture, packages provide the binding that holds the aspects of an object together, and give a very powerful mechanism for developing programs by a team of people. Packages are somewhat harder to understand than tasks, because they do not have a pre-defined purpose: packages are intended to encapsulate application-specific concepts. A package is a collection of declarations (of any kinds), written so that there is a clear separation between the specifications of interfaces and the bodies of program units that implement them. Packages are the basis for team work and program libraries in Ada, with the package interface used to ensure consistency between those units that use the declared entities and the package bodies that implement them.

Ada does not impose particular design rules, other than logical consistency, and permits the full variety of approaches to programming: top-down, bottom-up, back-to-front etc., including the object-oriented approach. Packages feature strongly in all of these. Because of the orientation of this course, I will concentrate on styles with an object orientation (i.e. not like Fortran COMMON or library subroutines). The basic idea is the abstract data type, which is expressed in Ada by the notion of privacy. A package specification gives just the information which the user needs to know, suppressing all details of implementation (which are given in the package body).

An abstract data type is defined not by its representation (structure and constraints), but by the operations that can be carried out on values of that type, and the relationships between them. Mathematically, an abstract data type is defined by the signatures of all the operations available for it, and the axioms, or rules of equivalence between these operations. Ada does not go quite as far as this, since there is no way of determining whether a given set of axioms is consistent and complete. Ada allows one to introduce a type (said to be private) without exposing its representation, in a package together with a set of procedure and function specifications for the operations on data values of that type. The procedure and function specifications give the signatures of the operations (that is, the types and modes of the parameters).

Using this style of programming (explained in more detail by Grady Booch, 1983), a package specification can be considered as a contract: it defines what can happen to objects of the designated type. Other program units may use objects of that type by nominating the package in their context clause; the package body contains all the implementation details.
Subprograms and packages may be written with compile-time parameters (including types and internal procedures), giving a kind of macro facility, known as generic program units. They may be compiled separately, but compilers must ensure that the type-consistency rules are applied even between separately compiled units.

1.5 Inter-dependency between program units

Ada uses a more flexible system of scope rules than previous block-structured languages. Each package specification introduces a set of identifiers, which may be used in any other program unit that mentions the package in its context clause. Thus the context clause at the head of each compilation unit identifies the other packages on which it depends. This permits both project management in the development of the program, and automatic consistency checking when the program units are submitted for compilation.

If a program unit is changed during the course of program development, it is automatically checked when it is re-compiled. Any packages it depends on are brought into scope, so that entities specified in them are visible. If the unit is a package body, then there is also a check that it provides implementations for everything specified in the package specification. If it is a package specification that has been changed, then every unit dependant on it must subsequently be re-compiled, to make sure that all uses of entities specified in the package are still consistent with their declarations.

This technique of controlled dependency has been found to give dramatic improvements in the productivity of programmers, particularly at the notorious integration stage. The number of errors that still remain when the units are linked together is very much smaller than with previous languages, as they have been detected earlier. Much useless work is thereby avoided.

1.6 Some misunderstandings clarified

Work with Ada has clarified a number of common misconceptions about programming: the status of coding, the relationship between a specification and a design, and the role of representations in performance trade-offs. It also emphasises the difference between a language and a compiler.

Coding is often considered to be the last and relatively trivial (but error-prone) stage in the construction of a program, after the architecture and design have been carried out. The misconception is about the separation of responsibilities between people and computers. Usually coding is taken to be the formulation (in a programming language) of the constructs that are converted by the compiler into executable instructions. In Ada this is less than half the truth, and gets the emphasis wrong (see Tooby 1986). The word 'coding' has the connotation of an information-preserving transformation from the comprehensible into the incomprehensible. This is entirely absent in Ada. Even the choice of representative values for enumeration types (necessary coding in most other languages) is avoided in Ada. The only sense in which coding has any meaning is as expression in a formal notation, which with Ada starts much earlier than the stage of writing executable instructions - indeed, it starts as soon as anything about the architecture (sets of package specifications) and design (algorithms and data structures) is recorded.
The relationship between the specification of requirements or intended effects and a design to achieve those effects is quite subtle. The design is not just added detail, but a different level of abstraction. Effects can be specified solely in terms of the states of the system before and after an action, whereas a design prescribes a selected arrangement of actions that together achieve the particular effect (see Pyle 1984). Ada allows one to describe designs (that is, ways of achieving required effects) without explicitly stating what the intended effects are. The specifications in packages are purely syntactic, and give no information about the semantics of the entities concerned.

Representations of data types have already been mentioned in connection with program portability. Ada requires the programmer to choose the representation for a private data type (where the space/time trade-off could be important) but permits automatic choices by compilers when there are no performance implications. But the representation choices are always separate from, and subservient to, the logical properties of the entities concerned.

The language Ada exists independently of any compiler for it. One of the benefits of the strong sponsorship for the whole Ada programme has been the setting up of compiler validation facilities to enforce the standard language. A compiler does the transformation from the comprehensible (Ada) program to the version that is not directly comprehensible to humans but is directly executable by a computer. Ada is not defined by the compilers but by the printed reference manual.

1.7 Conclusion

Ada is now a strongly-based international standard, with powerful political and industrial support. All the major manufacturers have compilers for it exist for their computers. It covers the technical requirements for embedded-computer systems, but is not limited to them: the techniques of fault tolerance and parallelism are generally required in information processing. In spite of its military origins, its importance is becoming widely recognised in civil, industrial and scientific computing.

Ada is the most important language for imperative compiled programming in the 1980s and 1990s.

2. Developing Ada Programs

Ada was the first programming language to take account of the fact that programs are developed by teams of people rather than by individuals, and to presume a computer-based working environment for programmers: the Ada Programming Support Environment or APSE. Program development in Ada is expected to be assisted by the use of software tools, with an automatically controlled collection of program parts, and consistent human interfaces for accessing them. This idea has subsequently been extended to apply to all aspects of a project, as an Integrated Project Support Environment, or IPSE.

It is not intended that the support environment will be identical to the target computer system that the program is being developed for: host/target working, with cross-compilation and remote debugging, are the norm.

2.1 Programming methodology

During the same period as Ada's growth, there has been a great upsurge of interest in programming methods: the ways in which programs are
developed. Methodology (the study of methods) allows us to see their similarities and differences. All are concerned with the progressive acquisition of information needed for successful completion of a project, and differ in the order this information may be obtained, the form in which it is expressed, and the checks that are applied to detect problems. Ada provides a consistent notation in which much of the information may be expressed, and extensive checks (by language rules enforced by compilers), but gives little guidance on the order in which information should be acquired. In this lecture we will discuss some ways of giving the necessary guidance, for use with Ada.

One thing is certain: whatever the method used and order chosen, there will sometimes be changes and revisions as development progresses. Thus the central feature of a support environment is a means of recording dependencies and controlling the versions of a program under development. Indeed, a library of program units must be handled by any Ada compiler, and any practical support environment must provide version and configuration control. An APSE may be considered as a system for manipulating program units, in other words one in which the objects handled are the constituent parts of the program under development.

2.2 Traditional methods

Algorithms have traditionally been sketched using a mixture of prose, mathematics and high level languages. While the result is an improvement over unconstrained prose, it is still weak and error prone (as well as ignoring the role of data types). The intention is to avoid premature commitment to detail by using phrases in natural language for subsidiary parts of the algorithm. Ada allows this kind of information to be formulated precisely, with no more detail specified than is desired, but with all the subsidiary constituents distinguished and categorised: suitable for further development (top-down) of the algorithm.

Subroutine libraries provide collections of useful pieces of program that are available to form the base for bottom-up program development. Their manner of use and calling sequences are traditionally given by example and prose descriptions. Ada again strengthens and tightens the structure by formalising more information: the package interfaces, where the library units form the corresponding package bodies. (It is not even necessary for the package bodies to be in Ada for this advantage to be obtained: the specification in Ada can give the calling information for checking when the package is used, even if the bodies are in Fortran or assembler.)

Fortran COMMON blocks are another traditional design aid: collecting together all the variables on one subject (that any of the subprograms may use or change). Ada packages may be used in just the same way, but with better separation of concerns and stronger checking. The data declarations (variables, which can include initial values, and constants, corresponding to BLOCK DATA) can be collected together into a package. Then whenever the data objects are involved by another program unit, it is sufficient to mention this package in the context clause. The declarations in the package are automatically incorporated - there is none of the problem of repeating all the COMMON statements (with possible clerical errors) in each subprogram. This must be considered as only a small advance, however, because there is no greater logical coherence in such an Ada package than in the corresponding COMMON block; it is solely the skill of the programmer that determines it, with no assistance or rule checking by Ada. For that we must look to new program design methods.
2.3 New Methods

The idea of the computer program being a model of some aspect of the real world has stimulated several design methods, which focus attention on relevant properties of the real world objects, and their dynamics. Abstract data types and object-orientated programming illustrate this style. Another important influence has been the mathematical study of program correctness, and formal specification of programs (as the reference for determining correctness). This puts attention to pre- and post-conditions, and invariants during program execution.

Note that the flexibility of Ada works both ways: because the language permits various methods of software design, it does not itself contain rules that enforce any particular method. (For this reason Cook 1986 claims that Ada meets none of the criteria for being an object-orientated programming language. Supplementary guidance is needed, as all authors point out.) We show in the following sections how to use Ada in each of these styles - so that the best of them can be used when circumstances are appropriate.

2.4 Abstract Data Types

The abstract data type approach ignores (initially) how a data type will be represented, and brings together all the operations related to objects of a particular type. In a way, it is the fact that the same operation can be applied to distinct objects that causes us to categorise them as of the same type. The operations on objects of a given type will include several that make use of values of particular characteristics of the object (which might later be used to guide the choice of representation), some that change values of characteristics, at least one that establishes for the first time a new instance of an object of the type, and perhaps one that destroys or removes from subsequent consideration such an object. Any of these may involve characteristics of other objects, or various characteristics of the same object.

In Ada, a package would be defined for each type of object involved, with a private type declaration (which allows users of the package to declare data objects of that type without being explicitly aware of their structure), and procedure or function declarations for all the operations concerned. These would typically have at least one parameter of the current type, and possibly other parameters of the same or different types, for interacting objects. There would be no direct counterpart in Ada of the axioms of a mathematical abstract data type: the semantics of the operations would be conveyed informally by their identifiers and supporting comments. Developing such a design would consist of choosing a representation (taking account of the relative frequency of the operations) and writing the package body.

2.5 Object-Oriented Design

Object-oriented programming goes one stage beyond this, by treating the objects as dynamic rather than static - in other words, not only do their values change by explicit operations, but progress spontaneously and autonomously. The structure to use for this in Ada is significantly different from that for an abstract data type: here we need a packaged task.

We introduce a package for each object (in the object-oriented sense), of which the principal part is a task specification containing entries for the
various events in which the object may be involved. The corresponding
package body contains the task body which models the dynamic behaviour of
the object between events, the interaction at each event being defined by
the rendezvous at the corresponding accept statement. It is usually the
case that events are partially sequential, but predominantly in a group
where any of a set of events may occur next - the structure of a select
statement used to show non-determinism in an Ada program.

Since there are likely to be a number of objects of the same type, Ada has
the concept of a task type, so that the task specification corresponds more
to the data type in a package for the abstract data type style of
development. The rest of the program can then gain access to these
facilities by the usual Ada method of naming this package in context
clauses, and then making declarations for objects of the types specified.
Object-oriented programming in Ada leads to data objects of task types.

2.6 Formal Specification

Formal specifications of program units concentrates attention on effects
rather than ways of achievement. The total effect of an ordinary procedure
is to change the state of the system in some way: its formal specification
is a mathematical specification of the relationship between the two states
(i.e. the values of all the relevant variables or characteristics of relevant
objects in the system) before and after an execution of the procedure. This
relationship is not necessarily the same as an algorithm (and in general
can be expressed in an algorithmic form only in the most simple cases): a
program design consists of finding a way of satisfying the relationship,
and a proof of correctness is a demonstration that a given program does
cause the stated relationship to be established. There is certainly no
presumption that any such method might be unique, or even any necessity
that it should even exist.

Formal specifications also concentrate attention on state invariants - that
is relationships between characteristics that remain the same while the
state changes. The properties of data types are like this, and input/output
operations can be expressed as actions to maintain some invariant between
the state in the real world outside the computer and the state of the model
inside it.

In Ada, it is possible to express these state relationships (preconditions,
postconditions and invariants) only in the simple cases corresponding to
propositional calculus: more complicated relationships require predicate
calculus, with existential and universal quantifiers, which are beyond Ada's
expressive power. Thus supplementary notations are needed for these, but
Ada programs can be annotated with these to show the intended effects and
formal properties required.

2.7 Consolidation

During the few years since Ada has been available, various design methods
have been investigated (including MASCOT, Jackson 1986, and Jackson's
Structured Development) but the most prominent has been the object-oriented
design method of Grady Booch. However, it must be made clear that there is
no guarantee of success with this method (any more than with any other);
software design still depends fundamentally on the skill and understanding
of the software engineer.

Baker (1986) contrasts the way that Ada encourages the use of design
solutions that are appropriate to the requirements of the intended system,
with the necessity - pre-Ada software designs to consider carefully the limitations of the implementation language in order to avoid implementation difficulties.

Michel (1986) showed that an object-oriented design technique can be used successfully to design Ada systems exhibiting enhanced modificability, efficiency, reliability and understandability.

2.8 Conclusion

Objects within an Ada program include variables and tasks, with types and associated properties that reflect the characteristics of the real world objects that are modelled by the software. A program itself is an important subject particularly considering the development process, in which program units are objects of concern. The rules of Ada, both for the relationship between data objects during program execution, and between the compilation units of the program during development, ensure logical consistency and coherence, which are supported by software tools and facilities in the Ada Programming Support Environment.

* * *

REFERENCES


Tooby B.: Against the use of coding; Ada User (July 1986).
MODULARIZATION AND REUSABILITY IN CURRENT PROGRAMMING LANGUAGES

Paul Klint

Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, University of Amsterdam, The Netherlands

How well do modularization constructs in current programming languages allow the construction of truly reusable modules? This question is answered by examining the implementation of the datatype queues in Pascal, Modula-2, Ada and Smalltalk. The merits of object-oriented languages versus algorithmic languages are discussed from the perspective of reusability.

1986 CR Categories: D.2.2 [Software Engineering]: Tools and Techniques - Modules and interfaces, Software libraries; D.3.3 [Programming Languages]: Language Constructs - Modules, packages.

1980 Mathematics Subject Classification: 68B10 [Software]: Analysis of Programs - Semantics.

Key Words and Phrases: software engineering, reusable software, modules, abstract data types, object-oriented programming.

1. INTRODUCTION

Over the last two decades it has become widely recognized that abstraction is the key technique that allows software developers to control the ever increasing complexity of the software systems they are building. The primary concerns are to control the quality (conformance to specifications, reliability, maintainability) and costs of the resulting product. The quality of a software product is determined by the quality of all steps in its production process, ranging from requirements analysis, problem specification and initial program design to coding, testing and maintenance. The costs of a software product are also determined by the costs of all steps in its production process. More often than not, the objectives of maximizing quality and minimizing costs cannot be reconciled.

A common abstraction technique is to decompose a program into a number of modules and to distinguish between the behaviour of the module as it can be perceived from the outside, and its internals, i.e. the details of the implementation that realizes its external behaviour. The advantages of this form of information hiding are manifold. The interfaces between modules can be minimized (thus controlling the number of inter-module dependencies) and the implementation of modules can be changed or optimized without affecting the users of these modules.

A common technique for reducing the costs of software products is to reuse parts of previously written and tested programs. This technique is by no means new. In the mid-fifties, one noticed already that many programs require common operations such as computing the cosine function, inverting a matrix, sorting a file or computing a standard deviation. The programmer of each of these operations needs to have expertise in a particular area. The user of these operations might not have this expertise. This observation has led to the construction of many subroutine libraries in specialized fields of application. In order to accommodate the needs of as many future users as possible, the operations provided by these libraries tend to be very general and highly parameterized.

Clearly, the modules of a modularized program also lend themselves to reuse. But which properties should the module constructor in a programming language have in order to maximize the possibilities of reuse? As a general rule, which will be detailed in the following sections, the possibilities of reuse of a module are maximized if the assumptions about its use are minimized. This can be achieved by delaying, as much as possible, the moment that a module is bound to its
"users" (i.e. other modules). Contrasting this with the desire to bind such information also as early as possible—if one wants to perform static (compile-time) typechecking in order to achieve run-time efficiency—we have concisely formulated one of the many dilemmas a language designer has to face.

I will discuss modularity and reuse issues in two families of current programming languages: the "algorithmic languages" and the "object-oriented languages". The major distinction being that the algorithmic languages stay within the framework of statically typed languages and have limited capabilities of reuse, while object-oriented languages use dynamic typing and lend themselves to more general reuse.

This paper is organized as follows. In section 2 a common example is introduced that will be used throughout. In section 3 the algorithmic languages Pascal, Modula-2 and Ada are discussed. Section 4 discusses object-oriented languages in general, and Smalltalk-80 in particular. Section 5 contains some concluding remarks.

2. A COMMON EXAMPLE

As a common example I will use the queue data type. A queue is a first-in-first-out list of elements on which the following operations are defined:

- **empty**: true if there are no elements in the queue, false otherwise.
- **full**: true if no more elements can be added to the queue, false otherwise.
- **insert element**: insert an element at the rear of the queue.
- **delete element**: delete an element from the front of the queue; return that element.

![Figure 1. The queue data type.](image)

Figure 1 illustrates these operations. There are two obvious implementations of queues: by means of a linked list and by means of a circular buffer. The former allows the implementation of unbounded queues, the latter provides a more compact representation since no explicit links between the elements in the queue have to be stored. See [Knuth68] for a discussion of these alternatives. The entities needed for the circular buffer implementation (also see figure 2) are:

- **MaxQsize**: maximal number of elements in a queue.
- **elems**: array containing the queue elements.
- **first**: front of the queue.
- **last**: the rear of the queue.
- **size**: the number of elements in the queue.

In this paper we are interested in the following three questions:

1. Is it possible to define the data type queue in such a way that the implementation can be changed (e.g. switching from a circular buffer to linked lists) without affecting the users of the data type?
(2) Is it possible to define the data type queue once, such that queues with elements of a different type can be defined using that definition?

(3) Is it possible to define queues in which the elements are heterogeneous, i.e. a queue may contain elements of different types?

3. ALGORITHMIC LANGUAGES

3.1. Pascal

Pascal [Wirth74] is a widely used general purpose programming language. It is a descendant of Algol60 [Naur63]. Most innovative were, at the time of its design, the facilities for defining datastructures (records) and the static (compile-time) typechecking of programs using them.

The Pascal implementation of queues is shown in figure 3. Note how the wrap-around of the indices first and last is achieved using the mod function. The consequence of this choice is that indices of queue elements range from 0 to MaxQsize-1, and the last element of elems remains unused (it is not possible in Pascal to declare arrays with constant expressions as bounds such as, e.g., MaxQsize-1). The procedures underflow and overflow are left unspecified.

The problems with this Pascal implementation are three-fold:

P1: Implementation hiding: the representation of queues by means of an array is visible and can not be changed transparently to another representation using, for instance, linked lists. It is even impossible to guarantee the consistency of this implementation, since users of the queue may freely change elements of record Queue.

P2: Generic types: the type of the queue elements is fixed and the code as shown in figure 3 (which implements queues of integers) has to be repeated for each desired element type. The only true modification required in each case is the substitution of the desired type for the type ElemType.

P3: Heterogeneous types: it is impossible to implement heterogeneous queues. A possible implementation using variant records for a queue with elements of type boolean, integer and real is shown in figure 4. This approach has the disadvantage that the type definition of the variant record that lists all allowed types has to be edited each time a new type is added. This becomes very bothersome when generic operations on queues are defined such as, for instance, a print procedure (see section 4.3 for a further discussion of this topic).
const MaxQsize = 100;
type ElemType = integer;
  Queue = record
    first, last, size : integer;
    elems : array[0..MaxQsize] of ElemType
  end;

procedure initialize(var Q : Queue);
begin
  Q.first := 0; Q.last := 0; Q.size := 0
end; {initialize}

function empty(Q : Queue) : boolean;
begin
  empty := Q.size = 0
end; {empty}

function full (Q : Queue) : boolean;
begin
  full := Q.size = MaxQsize
end; {full}

procedure insert(var Q : Queue; E : ElemType);
begin
  if full(Q) then overflow
  else
    begin
      Q.size := Q.size + 1;
      Q.last := (Q.last + 1) mod MaxQsize;
      Q.elems[Q.last] := E
    end
end; {insert}

procedure delete (var Q : Queue; var E : ElemType);
begin
  if empty(Q) then underflow
  else
    begin
      Q.size := Q.size - 1;
      Q.first := (Q.first + 1) mod MaxQsize;
      E := Q.elems[Q.first]
    end
end; {delete}

Figure 3. Queues in Pascal.

types ElemTypes = (boolean, integer, real);
  ElemType = record case kind : ElemTypes of
    boolean : (boolvalue : boolean);
    integer : (intvalue : integer);
    real : (realvalue : real)
  end;

Figure 4. Heterogeneous queue implemented with variant records.
3.2. Modula-2

Modula-2 [Wirth85] is a descendant of Pascal and is intended for systems programming in general, and multiprogramming in particular. The language extends Pascal in several ways. Here we are only interested in the module concept. Modules are defined in two separate parts: a specification part defining the external interface of a module and an implementation part defining its implementation. This distinction provides not only proper implementation hiding of modules, but also forms the basis for separate compilation in the Modula-2 implementation.

The implementation of queues in Modula-2 is given in figure 5. Note that only the name of the type Queue and the name and type of the five operations defined on queues are specified in the definition module. All information concerning the representation of queues and the algorithms used for implementing the five queue operations is hidden in the implementation module. The type Queue is defined as a pointer to a QueueRecord (which contains the actual information related to the queue). This construction is necessary to circumvent the constraints imposed on the "opaque" export of a type, which only exports the name of a type but not its representation.

The advantage of the Modula-2 implementation is that a true division has been achieved between the specification of a module and its implementation, thus eliminating problem P1 (implementation hiding). However, the Modula-2 implementation shares problems P2 (generic types) and P3 (heterogeneous types) with the Pascal implementation. Modula-2 also permits the (unsatisfactory) solution of P3 based on variant records.

3.3. Ada

Ada is based on a language designed by J. Ichbiah (Cii Honeywell Bull) in 1977. After several revisions and international reviews, the reference manual of the resulting language "Ada" was published in 1980 and the definition was finilized and accepted by ANSI in 1983 [DoD83].

Ada supports the concept of separating the definition and the implementation of a module in a comparable but more general way than Modula-2. Modules are called packages in Ada terminology and definition part and implementation part are called package specification and package body, respectively. However, Ada extends the notion of modules in a major way, by allowing generic packages. A generic package is a template for a package, but it is not a package in itself. Generic packages can best be compared with compile-time macros which exist in many languages: a macro definition consists of a macro-name, some formal parameters and a replacement text possibly containing these parameters. Every occurrence of the macro in the text of a program is then replaced—at compile-time—by the replacement text of the macro definition after proper substitution of the parameters. In a similar way, generic packages are instantiated by properly substituting the parameters of the generic package. Generic packages form a mechanism for delayed binding, i.e. the "tailoring" of the generic package is performed when the package is instantiated.

The Ada implementation of queues is given in figure 6. This implementation is inspired by an example given in [Hibbard, Hisgen, Rosenberg, Shaw & Sherman 81]. In this case all initializations of the queue are performed in the declaration of the type Queue. The type Queue is declared limited private; this forbids assignment operations and equality tests for queues. An instance of the generic package Queue can, for example, be obtained by the following declaration:
definition module Queue;
  type ElemType = integer;
    Queue;
procedure initialize(var Q : Queue);
procedure empty(Q : Queue) : boolean;
procedure full(Q : Queue) : boolean;
procedure insert(var Q : Queue; E : ElemType);
procedure delete(var Q : Queue; var E : ElemType);
end Queue.

implementation module Queue;
  from storage import allocate;
  from system import tsize;
  const MaxQsize = 100;
  type QueueRecord = record
    first, last, size : integer;
    elems : array [0 .. MaxQsize-1] of ElemType;
  end;
  Queue = pointer to QueueRecord;
procedure initialize(var Q : Queue);
begin
  allocate(Q, tsize(QueueRecord));
  Q^.first := 0; Q^.last := 0; Q^.size := 0
end initialize;
procedure empty(Q : Queue) : boolean;
begin
  return Q^.size = 0
end empty;
procedure full (Q : Queue) : boolean;
begin
  return Q^.size = MaxQsize
end full;
procedure insert(var Q : Queue; E : ElemType);
begin
  if full(Q) then overflow
  else
    Q^.size := Q^.size + 1;
    Q^.last := (Q^.last + 1) mod MaxQsize;
    Q^.elems[Q^.last] := E
  end
end insert;
procedure delete (var Q : Queue; var E : ElemType);
begin
  if empty(Q) then underflow
  else
    Q^.size := Q^.size - 1;
    Q^.first := (Q^.first + 1) mod MaxQsize;
    E := Q^.elems[Q^.first]
  end
end delete;
end Queue.

Figure 5. Queues in Modula-2.

package intqueue is new queue(ElemType => integer, MaxQsize => 100);

The result is a package declaration, defining queues of at most 100 integers. Ada allows us to hide the implementation of queues and also to define generic types. However, Ada does not solve P3 (heterogeneous types), in another way than using variant records as can be done in Pascal and Modula-2.
Figure 6. Queues in Ada.

4. OBJECT-ORIENTED LANGUAGES

4.1. What are object-oriented languages?

Algorithmic languages are based on a computation model that distinguishes between operands (i.e. data values of a certain named type) and operators (i.e. functions or procedures operating on operands of certain types and producing a typed result). Operands are thus passive while operators
are active: operands represent the current global state of a computation while operators can transform this global state. It is essential that the context in which a certain computation is to be performed is responsible for the selection of a properly typed operator to carry out the desired computation. If, for instance, an addition operator exists for types integer, real and complex, one of these three operators has to be selected at each position where an addition operator is used.

The so-called object-oriented languages are based on a different computation model. The entities considered in this model are called objects which may have their own private storage (containing data values). The notion of computation is captured by the concept of message passing: each object is capable of answering certain messages. A message is a request to the object receiving it and the latter has the obligation to provide a method (cf. operator) for answering it. During the construction of the answer the private memory of the object may be changed and messages may be sent to other objects. It is essential in this model that the sender of a message does not know which method will provide the response to the message. In this way object-oriented languages provide data abstraction and--as we will see below--reusability.

Usually, many objects have equal properties, i.e. they respond to the same messages in the same way. It is common to organize those objects into classes. An element of a class is called an instance of that class. The advantage of this organization is that the methods needed for answering messages do not have to be specified separately for each object; they can be specified once for a whole class.

The notion of classes can be refined by relating classes in order of increasing specificity. A more specific class (e.g. automobiles) inherits properties (e.g. has a number of wheels) from a more general class (e.g. vehicles), but it can also add specific properties (e.g. consumes fuel). The more specific class is called a subclass of the more general one. In this way, a subclass can extend or modify an existing class by specifying the differences with it or by specifying new messages that are typical for the subclass.

A distinction should be made between an object-oriented language and object-oriented programming. The former is a programming language which is strictly based on the message/object computation model (see next section). The latter is a style of programming that simulates the message/object model in an existing programming language (see section 4.5). See [Cox86] for a general introduction to object-oriented programming.

4.2. Smalltalk

The language Simula [Dahl, Nygaard & Nygaard71] is a statically typed superset of Algol60. Simula introduced the notions of classes and of class inheritance (called class concatenation in Simula) in the context of solving simulation problems. These two concepts combined with the notion of dynamic binding stemming from Lisp form the basis for Smalltalk [Goldberg & Robson83], which is nowadays the prime example of an object-oriented programming language. Some aspects of Smalltalk are now discussed by commenting on the Smalltalk implementation of queues (see figure 7). For a complete description of the language the reader is referred to [Goldberg & Robson83].

Each class has a name (e.g. Queue) and may be a subclass of another class. All classes are ultimately a subclass of the most general class Object, which defines general operations such as testing the class of an instance, and comparing and copying instances. The “private memory” of each object consists of a number of instance variables; each instance of a class has a separate set of instance variables.

The messages accepted by a class have a name and (optional) parameters. Parameters may be separated by user-defined delimiters. Sending a message to some class instance is written as the juxtaposition of the instance and the message.

Some examples of messages that might be sent to q (an instance of class Queue) are:

```
q empty
q insert: 3
```

(send message empty to q)

(send message insert: with argument 3 to q)
class name: Queue
subclass of: Object
instance variables: first last size maxsize elems

initialize: n
  first ← last ← size ← 0.
  maxsize ← n.
  elems ← Array new: n

empty
  ↑ size = 0

full
  ↑ size = maxsize.

insert: e
  self full
    ifTrue: [ ↑ 'overflow' ]
    ifFalse:
      size ← size + 1.
      last ← (last + 1) rem: maxsize.
      elems at: (last + 1) put: e

delete
  self empty
    ifTrue: [ ↑ 'underflow' ]
    ifFalse:
      size ← size - 1.
      first ← (first + 1) rem: maxsize.
      ↑ elems at: (first + 1)

Figure 7. Queues in Smalltalk.

Other examples are:

2 + 3
  (send the message + with argument 3 to 2; both 2 and 3 are instances of the class SmallInteger)

a at: 3 put: 'abc'
  (send the message at:put: with arguments 3 and 'abc' to a; if a is an instance of the class Array, this corresponds to the Pascal statement a[3] := 'abc')

a at: 3
  (fetch third element of a)

These examples show that some syntactic freedom can be achieved by using messages such as at: and at:put:. The parsing of these message patterns is done at compile-time.

The implementation of a message is called a method and is comparable to a procedure in a conventional language. Methods return a value to the sender of the message by means of the operator ↑. They may also refer to the receiving object itself (i.e. the class instance that is currently executing this method) by means of the pseudo-variable self. When a message is sent to an object, the selection of the method to be invoked is determined--at run-time--by the class of that object.

Control structures such as if- and while-statements are modelled by messages sent to the class of Boolean values. The if-statement, for instance, is provided by the message ifTrue:ifFalse:. The Smalltalk expression:

x > 0
  ifTrue: [ sign ← 1 ]
  ifFalse:[ sign ← -1 ]
corresponds to the following Pascal statement:

\[
\text{if } x > 0 \text{ then sign := 1 else sign := -1;}
\]

The semantics of the Smalltalk expression is as follows: the expression \( x > 0 \) is evaluated and yields one of the two possible instances of the class \textit{Boolean}: \textit{true} or \textit{false}. Next, the message \texttt{ifTrue:ifFalse:} is sent to this \textit{Boolean}, which in turn responds by evaluating either the first or the second argument of that message.

The Smalltalk implementation solves all three problems we have posed in section 2. Problem P1 (implementation hiding) is solved by the class mechanism itself. Problems P2 (generic types) and P3 (heterogeneous types) are solved by run-time method selection which avoids nearly all static constraints on the types of the elements in the queue. I will further discuss this issue in the next section.

4.3. The effect of object-oriented programming on program organization

In languages based on the operator/operand model, programs are organized around the definitions of the operators required to solve a particular problem. In languages based on the message/object model, programs are organized around the definitions of classes; the definitions of operators are distributed over the class definitions. An example will illustrate this.

Suppose we want to add a print operation to a heterogeneous queue data type. Assume that values of types \( t_1, \ldots, t_n \) may occur as elements in the queue and that these values can be printed by the procedures \( t_1\text{print}, \ldots, t_n\text{print} \). In a Pascal implementation using variant records (see figure 4), a print routine for queues will have the following general structure:

```
procedure print(Q : queue);
begin
  for each element E in Q do
    case E.kind of
      t_1: t_1print(E.t_1value);
      ...
      t_n: t_nprint(E.t_nvalue);
    end;
  end print;
end;
```

One can clearly see that this procedure has to be aware of the various types the queue elements may have. Whenever elements of another type are allowed to appear as elements in the queue, a new entry has to be added to the above case statement.

The same problem formulated in Smalltalk-style will have the following structure:

```
print"print method of class Queue"
  self do: [ :e | e print ]
```

This can be paraphrased as:

```
print"print method of class Queue"
  for each element E in Q do
    send print to E
```

In this case the selection of the precise code to be executed in response to the message \texttt{print} is not made in the print routine for queues, but is determined by the class of which each queue element is an instance. The code of the above print method thus becomes independent of the types of the elements in the queue.
This form of program organization is also known as *data-driven programming* and leads to easily extensible implementations. Another example may further illustrate this. In a conventional compiler for some language \( L \), a source program is converted into an abstract syntax tree and this tree is processed in several phases. Typical operations to be performed are: *check* (check types), *dataflow* (perform dataflow analysis), *allocate* (allocate registers) and *generate* (generate code). Each of these operations will be implemented as a procedure operating on the complete syntax tree and knowledge about the various constructs that may occur in it has to be repeated in each of these four procedures. This same problem can be implemented in an object-oriented style by defining a class for each kind of statement in \( L \). Each such class has to implement methods for the four desired operations. This leads to a totally different organization: all information about one language construct is concentrated in one class definition. This approach is superior over the traditional one, if language \( L \) is not fixed (i.e. \( L \) is still under design and language features may come and go), or if several dialects of \( L \) have to be implemented and one wants to maximize the sharing between these implementations. See [Veen86] for a description of this technique in the context of the SUMMER programming language [Klint85], and [Abelson, Sussman & Sussman85] for a description in the context of Lisp.

### 4.4. Disadvantages of object-oriented languages

The advantages of object-oriented languages for reusability and extensibility will by now be clear; what disadvantages do these languages have? First, the message/object model forces an asymmetric view on operations which are inherently symmetric such as, for instance, arithmetic operators. The problem then arises how conversions can be inserted in mixed-mode expressions such as, e.g. an addition with operands of types integer and real. Straightforward application of the view that "\( 2 + 2.5 \) is evaluated by sending the message + with argument 2.5 to the integer instance 2" automatically leads to selection of the addition operation on integers, which is clearly undesired. In this particular case, the problem can be solved by assigning a level of generality to all classes defining arithmetic operators and by defining coercion protocols between them: when a message is sent to a less general class with an argument of a more general one, the former should be coerced to the class of the latter and the message should be re-sent to the (coerced) class instance. No solutions exist for this phenomenon which are both elegant and general.

Secondly, not all problems can be modularized by means of strictly linear inheritance chains as provided by, for instance, Smalltalk. A typical example is the class *waterplane*, which could best be defined as a subclass of the classes *boat* and *plane*. Here one needs a form of *multiple inheritance*.

Finally, there is a certain overhead associated with run-time method selection. Measurements show that the operation of sending a message to an object is roughly twice as expensive as performing a procedure call [Cox86].

### 4.5. Object-oriented programming in existing languages

It was already mentioned in section 4.1 that object-oriented programming techniques can, in principle, also be used in conventional programming languages. In this section I will briefly review the language features that are required to support such an object-oriented style. These features are: *information hiding, generic operations, dynamic binding, inheritance, and automatic storage management*.

Clearly, without an information hiding mechanism, it is impossible to implement a notion of "objects". Modules from Modula-2 and packages from Ada are acceptable from this point of view.

As we have seen in the preceding sections, it frequently occurs that messages with identical names are defined for several classes. Such messages are *generic* (or *polymorphic*), i.e. their behaviour depends on the type of object to which they are sent. Such operators are not expressible in Modula-2 (since overloading of names is forbidden) and only to a limited extent in Ada (since the language is defined in such a way that all overloaded and generic names can be resolved at compile-time).
The late moment of binding of names to their definition (also called *dynamic* binding) is one of the major reasons for the flexibility of object-oriented languages. This aspect of object-oriented languages is very hard to represent in statically typed languages, but is present in many not object-oriented languages such as, e.g., Lisp.

The notion of linear inheritance can be mimicked by the *import* and *use* constructs in, respectively, Modula-2 and Ada.

Strictly speaking, automatic storage management is a property of a programming language implementation, rather than a language feature in itself. Availability of automatic storage management, however, makes implementations of datatypes more reusable since details concerning allocation and deallocation can be omitted. If we consider, for instance, an implementation of queues based on linked lists, the question arises whether a value just deleted from the queue should be deallocated or not. It is most likely that the implementor is forced to leave this decision to the user of the queue data type. But by doing so, he also compromises the implementation independence of his data type.

This discussion makes clear that the possibilities for object-oriented programming are absent in the case of Modula-2 and that they are limited in the case of Ada. What are the possibilities in other languages? I will briefly mention two cases: Lisp and C++.

Clearly, Lisp scores high when checking the features mentioned above. It is therefore not surprising that various Lisp extensions have been defined that provide primitives for object-oriented programming. The functionality of these systems is comparable to or even encompasses that of Smalltalk. In particular, improvements have been made in the area of introducing multiple inheritance and of user-defined inheritance schemes.

C++ [Stroustrup86] adds the notion of classes to the language C [Kernighan&Ritchie78], in a similar way as Simula added classes to Algol60. C++ is a superset of C and supports inheritance ("derived classes") and operator overloading, but--staying within the framework of compile-time typechecking--it does not support general polymorphic functions.

In object-oriented languages all operations are performed by means of message passing, independently of the amount of work required to compute them. For low-level operations such as, e.g., the addition of two numbers, the overhead of message passing will be relatively large. Object-oriented programming in a suitable conventional language has then distinct advantages since such low-level operations can be performed by direct procedure calls thus eliminating the message passing overhead.

5. CONCLUDING REMARKS

We have discussed two families of languages, each with different properties as far as reusability is concerned. The algorithmic languages place a strong emphasis on compile-time typechecking: types are identified by a name and all procedures and data structures in a program are explicitly typed. At compile-time, this type information can be used to check that all expressions and statements in a program are properly typed. The advantages of this approach are reliability (all improperly typed expressions are detected at compile-time) and efficiency (there is no need to keep type information at run-time). The disadvantage is limited reusability due to the early binding of type information.

In object-oriented languages, types are not identified by name but by the set of operations that are defined on values of that type. This leads to very reusable code (each method only sends a limited set of different messages to its arguments and will therefore work for all types that implement these messages), but is less efficient (since run-time management of type information is required).

The distinction between compile-time and run-time typechecking is less absolute than is suggested here. By performing a sufficiently sophisticated analysis of programs one may infer type information that is only implicit in the text of the program. See, for instance, [Suzuki81] or [Boring & Ingalls 82] for an application of this idea to Smalltalk. In this way, one can eliminate most (but not all) run-time typechecks from a language that would otherwise require complete dynamic checking.
The best of two worlds can be obtained by using elastic typing schemes [Heering & Klint85], in which typechecking is performed as soon as type information becomes available (e.g., due to input operations) or can be inferred (e.g., by filling in details in an incomplete program).

ACKNOWLEDGEMENT

Ard Verhoog, Pum Walters and Freek Wiedijk made several comments on drafts of this paper.

LITERATURE


A. Goldberg & D. Robson, SMALLTALK-80, The Language and its Implementation, Addison-Wesley, 1983.


AN INTRODUCTION TO SYMBOLIC MANIPULATION

J. Vermaseren

NIKHEF-H, Amsterdam, The Netherlands

Introduction

In the early days computers were used for number crunching and data processing only. Nowadays with increased hardware availability fancier applications are being made many of them classified under the name "Artificial Intelligence". One of the subfields of AI is called symbolic manipulation, or the manipulation of formulae. The term computer algebra is also used in this context but is rather confusing as many mathematicians use it also for computer programs for group theory.

As far as the manipulation of formulae is concerned most of us have already some experience: Suppose you are editing a file and notice that at five places the string (a+b) occurs. You may decide to define c = a+b and let the editor replace everywhere (a+b) by c. At this point you have done symbolic manipulation. Actually the whole field of symbolic manipulation is built around substitutions like: replace "A" by "B". All other operations are either related to this, or built in versions of the replace, as we will see further on.

One may ask what this "symbolic manipulation" is used for? There are three main application fields:

/Al/: Common algebra. Usually the computer can do the operations much faster than the user himself. The great benefit comes however from the reliability. Also mending mistakes is much easier: change one line and resubmit.

/Bi/: Uncommon algebra. The manipulations may involve formulae that could fill a book. The results often find their way to a Fortran program for further processing. Rarely the results can be derived by 'hand'.

/Ci/: To gain insight.
Often the derivation of a 'simple' answer gives enough insight to prove it in a different way. Suppose you are faced with four equations of the type

\[ 371x_1^8x_2^2 + 1801x_3^7x_4^3 + 201 = 0 \]

and all coefficients are taken modulo 1831.
Finding a solution to these equations is hard and takes even some time by computer. Once the solution has been found it is very easy to verify that it is a solution.
Part I The Mechanics

Inventory

There are nowadays many symbolic manipulation programs available, so it is impossible for me to be complete when looking at them.
The most popular ones are (for physicists):

(a) Macsyma It runs on several machines: VAX, SUN, Symbolics. It is written in LISP and suffers from its restrictions when not running on the Symbolics machine. Of all these programs it can boast of the greatest amount of manyears that went into its construction.

(b) Maple A rather new program that runs on many systems like VAX, IBM VM/CMS, Microvax...

(c) Reduce It runs on VAX, Apollo, IBM... It is usually severely hampered in the size of the problems it can deal with by the LISP that it uses.

(d) SMP Also a rather new program written in C. It runs on many systems. One should use it with great care though, as its debugging is not complete yet.

(e) Schoonschip This program has been written in Assembler for CDC 66-bit machines and for the 68000 family. It is by far the fastest of these programs. Nowadays it even runs on home computers (Atari ST).

These programs can be divided into two classes:

/I/ The consumer class:

For these programs the emphasis lies on a wealth of instructions and built-in knowledge. It is assumed that the user has a rather standard problem which is not too big. The algorithms that are used must be rather general and correspond usually to a brute force method for which the computer is of course very suited.

/II/ Research class:

All necessary features are present to provide one's own knowledge. The program should be unhindered by buffer sizes etc. but be open-ended so that there is in principle no limit to the length of a "computation". The emphasis should lie on speed. The complexity of the problem is determined by the user. The biggest disadvantage of these programs is that the user has to think about how to solve his problem.

Of the 5 mentioned programs the first 4 are of the consumer class while the last one is of the research class. It is a very rare example of a general research class program as many research class programs are rather dedicated to one type of problems, like for instance the theory of finite groups.

In practice one can work best using a consumer class program like Macsyma for the common work and a program like Schoonschip for the more exceptional problems.
The parts of a symbolic manipulation program.

In this section we will see from what kind of pieces a symbolic program is made. This is mainly meant to separate the vital pieces from the peripherals. We have in principle 4 pieces:

(1) Input handling
   This concerns the reading of the input and the compilation or interpretation of it. Usually there is a hybrid of compilation and interpretation when jobs are run in batch.
   From our viewpoint this step is trivial as these routines merely define the syntax of the language, not the underlying principles.

(2) The "central algebra processor"
   This part contains the pattern matching routines, the substitutions, the term generators and the special instructions. It has access to a built-in knowledge base. This is the heart of the program.

(3) The sorting routines and the data storage
   When two terms are otherwise identical their coefficients should be added and one (or both if the result is zero) should be removed. Also the terms should be written in a unique representation e.g. \( ab+ba \rightarrow ab+ab \rightarrow 2ab \) if \([a,b]=0\). This requires a (usually) complicated subprogram to "normalize" terms and then some good sorting routines.
   At this point it depends on the program whether external sorting (disk files) is allowed so that only disk size limits the size of a formula.

(4) Output routines
   Output should be unambiguous and in a form that can be read again by the input routines. For the consumer class programs much work has gone into making very fancy outputs that write formulae like:

   \[
   z = \frac{x^2 y^2}{(12 a + 13 b)^4 (8 x + 5 a)^3}
   \]

   It looks very nice and may some day even be coupled to a wordprocessor. For a research program this kind of capacity is rather superfluous and merits a quote:
   "It is easy to be blinded to the essential uselessness of it by the sense of achievement you get from getting it to work at all"
   (Hitchhikers guide to the galaxy).

   Some programs will try to factorize the output if the user asks for it. This is very nice when it works.
   For large expressions it can become rather time-consuming however, so the best method here seems to have the user guess and let the machine show the result. One may call this "interactive guesswork".
   There is quite some mathematical research going on in this field so the situation may improve dramatically in the near future.
Fundamentals

In this section we take a good look at formulae in general. The basic unit of a formula is a term. A term is built up from subterms (of course everybody has his own terminology here). From here on the uniformity seems to end but this is mainly a matter of language. Whether an expression

\[ f = (a+b) \times (c+d) \]

is described in a LISP-like formulation

with

\[ f: (\ast, \text{piece1, piece2}) \]

\[ \text{piece1: (+, a, b),} \]

\[ \text{piece2: (+, c, d),} \]

or in a more direct way via

\[ f = \$1 \times \$2 \]

\[ \$1 = a+b \]

\[ \$2 = c+d \]

doesn’t change the result of the used operations.

One of the differences is that in the LISP-like way \(+, -, \ast, /\) are specific examples of general functions while in the second method they are worked out right away. Each method has its own advantages. Either way the result will be

\[ f = a \ast c + a \ast d + b \ast c + b \ast d \]

which is an expression (=formula) with 4 terms.

The types of variables that are encountered are:

1. Normal symbols
   e.g. \(x\) in \(f = 74x^2 + 76x + 2\).

2. Functions
   e.g. \(g\) in \(g(x,y)\).
   Some programs may use special ‘symbols’ for special functions like \(\sqrt{}\) or \(\sin\).

3. Vectors and indices
   e.g. \(p_{\mu}\), usually written as \(p(\mu)\).

4. Expressions
   They are formulae that are the objects of further manipulations, like \(f\) in
   \(f = (x+1)^2\).

5. Coefficients.

The main reason for differentiating between these types of objects is that it allows the automatic application of rules particular to the various kinds of variables.

The coefficients play a special role as they are not really variables. I do not know of operations like: replace all \(4^8\) by \(8\), etc. In general we like our results to be exact so the main number system to work in is that of the rational numbers. Their precision should in principle be unrestricted.
Sometimes the answer has the substitution of real numbers in it or the coefficients in a formula should be readable for a Fortran compiler so we may need a floating point representation.

The coefficients show a weakness of Schoonschip. It works internally in floating points with a 96-bits mantissa. There are however advanced techniques to convert floating point numbers to fractions so the user may only rarely be confronted with its limitations. The other systems work in principle in rational arithmetic but provide also a floating point capacity.

In practice the precision is always finite and one should moreover realize that computations become very slow when using fractions that have hundreds of digits. Often a modulus capacity is present. Many theories about numbers and polynomials work over finite number systems e.g. modulus 1831.

This gives

\[
\begin{align*}
1832 & \rightarrow 1, \\
\frac{1}{3} & \rightarrow 1221, \\
& \text{etcetera.}
\end{align*}
\]

The basic lay-out of a formula is:

Expression

\[ \downarrow \]

Terms

\[ \downarrow \]

Subterms, coefficient

If a subterm is a function:

\[ \downarrow \]

Arguments

Arguments are actually (sub)expressions

\[ \downarrow \]

......

etc.

The only thing this scheme does is to give a special status to the +/- operations by splitting an expression into terms and to the */: operations by splitting into subterms.

Example:

\[
t = 8ab + 7c(h(x + 2y, z + (a+b)^2))
\]
Substitutions

The main operation in symbolic manipulation is the substitution; i.e. replace one object by another. It is a two-step operation:
1. find a pattern in a term,
2. replace it by an expression and expand.

An example may be called for:

\[ f = 3a^4b^3 + \ldots \]
replace \( ab^2 \) by \( 5c + 3d + 9e \).

Wildcarding should also be possible (quantified variables):

\[ f = p(mu) \ast q(mu) \ast p(nu) \ast g(x,nu) \]
Substitute

\[ p(\tau(\rho)) = (\tau(\rho) + s(\rho))/2 \]
\[ q(\tau(\rho)) = (\tau(\rho) - s(\rho))/2 \]
or

\[ f = 12x^7 + 16x^6 + \ldots \]
Substitute \( dx \times x^n \times x = x^{n+1}/(n+1) \).

Special operations are usually substitutions that are generated by the knowledge base of the program. An example is the formula above, which might be generated by a symbolic program when the ‘integrate with respect to x’ instruction is given.

Another example is:

\[ f = c \ast b \ast d \ast a \]
in which \( a, b, c, d \) are non-commuting objects.

The operation: “a commutes with all other variables” is equivalent with the substitutions:

\[ d \ast a \rightarrow a \ast d \]
\[ b \ast a \rightarrow a \ast b \]
\[ c \ast a \rightarrow a \ast c \]
which allows the program to cause a unique ordering of the subterms.

For \( f = p(mu) \ast q(mu) \)
the operation: “write as dotproducts”
or “sum over all mu”
is equivalent to \( \tau(p(\mu)) \ast \tau(q(\mu)) = \tau \cdot \tau \)
and results in

\[ f = p \cdot q. \]

Some of these special operations are built in to improve speed as they are considered standard.

This includes some special very high-level operations like
SOLVE       Solve a set of equations,
DIFF     Differentiate,
INT      Integrate,
TRICK,TRACE Take traces of $\gamma$-matrices in High Energy Theory.

We will come back to these later.

**Term generation**

The generation of new terms takes place when (part of) a term has been substituted by an expression that may consist of several terms: If the user is not careful he may produce enormous amounts of terms this way. The main problem the program has to face is what to do with all these terms. There are basically two schemes: suppose we have a term in $f$:

\[ f = a_1 a_2 a_3 \]

and the substitutions

\[ a_1 \rightarrow b_1 + b_2, \]
\[ a_2 \rightarrow b_2 + b_3, \]
\[ a_3 \rightarrow b_3 + b_1, \]
\[ b_1 \rightarrow c_1 + c_2 + c_3, \]
\[ \vdots \]

The two actions are:

/A/ Substitute $a_1$. Collect all terms and see if there are cancellations or additions so that the next step can be started with as few terms as possible. Next substitute $a_2$ and collect terms etc. This is the linear approach.

/B/ Build a giant tree with all substitutions and collect the output only when there are no more substitutions or when the user asks for it.

![Diagram](image)

*etc.*

Method A is good when after each step the simplifications that occur make up for the extra overhead of the sorting of terms. There was a time most programs would work this way. The flexibility of method B
makes it nowadays the more popular method as the user could always ask for the sorting of the terms after each step:

\[
\begin{align*}
\text{subs} & \quad a_1 \rightarrow b_1 + b_2 \\
\text{sort} & \\
\text{subs} & \quad a_2 \rightarrow b_2 + b_3 \\
\text{sort} & \\
\text{subs} & \quad a_2 \rightarrow b_2 + b_3 \\
\text{sort} & \\
\text{etc.}
\end{align*}
\]

There exist spectacular examples in which the computation induces more than 100,000 different terms in the intermediate stages but in the end there are only very few different terms left. A wrongly timed sort instruction can cause the problem to be virtually insoluble.

We will conclude the first part of these lectures now with two “smaller” features that may be handy:

**Flow control**

Sometimes it is nice to have a set of instructions be executed more than once with various values for some parameters. Most programs are therefore equipped with a DO or a FOR instruction.

Also blocks of statements can be grouped together as a procedure that can be called, often even from a file. There are features to define substitutions that are tried at each stage, or only at the “point of definition”.

Not much use is made of if/then/else constructions. They would involve a mechanism of asking information about a term (or an expression). This can often be avoided as in:

- if (power of \(x > 10\)) discard term
- can be replaced by
- substitute \(x^{11} = 0\).

**Storage of results**

Larger projects need their partial results stored for further processing in later stages. This leads immediately to the question of how useful interactive work with a symbolic manipulation system is. When operating on a personal workstation there is usually not a big penalty for submitting jobs, go to the editor to make changes or additions, resubmit the job etc.

Especially for debugging this is a good way of working.

This last feature makes interactive work very hard for bigger projects. Actually the major research class program (Schoonship) used to have an interactive mode which was taken out again in the latest version. It was only used to demonstrate the solving of little problems.
Very often output is needed in a form that can be processed in numerical routines. For instance a very difficult formula has been derived and needs to be integrated numerically over a multi-dimensional space. This asks for a Fortran Monte Carlo program so the formula must be put in a Fortran program. All major systems offer the feature of giving the results in Fortran card image.

To conclude this section I would like to remark that the basic system to run symbolic manipulation programs with, seems to be a very powerful Personal Workstation equipped with a good editor. We will see some examples of their power in the sequel.

Part II. Examples of simple problem solving

In this section we will look at some specific examples of problem solving with a symbolic manipulation program. As we are much more interested in the underlying principles and the methods we want to use build in knowledge as little as possible, so the choice of manipulation program is not very critical. We have used here the program Schoonschip as it is available here at the school so you can try it out afterwards. In the end there will be a simple physics example for which Schoonschip will actually be the ideal program. Let us look now at the examples:

(1) Legendre Polynomials

Suppose you are interested in the expansion of a number of Legendre polynomials. They are defined via the reaction relation:

\[ P_n(x) = \frac{(2n - 1) x P_{n-1}(x) - (n-1) P_{n-2}(x)}{n} \]
\[ P_0 = 1 \]
\[ P_1(x) = x \]

How do we convert this into a program?

The program is presented in table 1. Here we will only explain the meaning of the statements. The line starting with A is a declaration of the (algebraic) variable x. Next are two lines with a Z that define expressions. These expressions are the start values \( P_0 \) and \( P_1 \). Keep indicates that these expressions are kept for the next step and *next finishes a step.

The instructions starting with a * indicate the end of a “module” and cause the program to start expanding the expressions, building a substitute tree, sorting the output and storing it away if asked for. The statements are read in module by module, then each module is compiled and run. So on the level of modules one deals with an interpreter, inside the module with a compiler.

Next is the DO loop over a number of modules with the formal parameter J. This parameter is immediately substituted if it is used between quotes as the ‘J’ in the next Z expression. The Z expression defines the recursion and the keep keeps the relevant expressions for the next step. The *end finishes the program.
\[ A \quad x \]
\[ Z \quad P(0) = 1 \]
\[ Z \quad P(1) = x \]
\[ \text{Keep} \quad P(0), P(1) \]
\[ \ast \text{next} \]
\[ \text{DO} \quad J = 2, 15 \]
\[ Z \quad P(J') = ((2*J'-1) \ast x \ast P(J'-1) - (J'-1) \ast P(J'-2))/J' \]
\[ \text{Keep} \quad P(J'), P(J'-1) \]
\[ \ast \text{next} \]
\[ \text{ENDDO} \]
\[ \ast \text{end} \]

\textbf{Table 1}

In reaction to this program Schoonschip prints the first 15 Legendre polynomials and it takes about 2 seconds which are spent mainly on the file manipulations around the keep instructions. It is of course not very illuminating to show the resulting formulae. They cover a full page.

\textbf{(II) Powerseries expansion}

Very often it is necessary to expand a formula in a given variable. This may be for numerical stability, for calculational speed or just to be capable to compute this formula at all. The example we will see here comes from the Macsyma examples and as a good example it serves no practical purpose whatsoever except to show the techniques involved. The task is to expand the formula

\[ f = a \cdot \sin(x^3) + b \cdot \frac{\log(1+x+x^2)}{x^5} \]

to terms in \( x^{30} \).

In Macsyma there is a built-in instruction that does this:

\texttt{TAYLOR(..., x, ..., 30)}

and this is all there is to it.

In our case we want to simulate this using Schoonschip. To this end we put the “knowledge” about the \( \sin \) and the \( \log \) in a library. What do we know?

\[ \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ... \]

\[ \log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - ... \]

How do we construct such an expansion?
For series expansion there is a special function DS that is an implicit sum.

Formally it is given by

$$\text{DS} [J, a, b, (f(J)), (g(J))] = \sum_{J=a}^{b} f(J) \cdot \prod_{k=a}^{J} g(k);$$

the final product must expand to a numerical coefficient and the coefficient of the first term is taken to be 1. In these terms the first 10 terms of the logarithm become:

$$\text{log}(1+x) = \text{DS} [J, 1, 10, (x^J), (-1/(J-1))].$$

We can make now a ‘subroutine’ which is named a block:

```
BLOCK    expan {x, J2}
Id        log1(x') = DS [J, 1, J2, (x^J), (-1/(J-1))]
Id        sin(x') = DS [J, 1, J2/2, (x^2*J), (-1/(2*J-1)/(2*J-2))]/x'
ENDBLOCK
```

The quotes indicate formal parameters of this ‘subroutine’ or block of statements. The new type of statements is here the Id or identify which indicates a substitution. This means that the program will try this substitute once when this statement is encountered. This block can be called by just referring to its name and giving it a set of parameters. We will see this shortly.

The next problem is that we do not need \(\text{log}(1+x)\) but \(\text{log}(1+(x+x^2))\). We can solve this by using \(\text{log}(1+y)\) and then replacing \(y\) by \(-x+x^2\). Alas things are not this simple as Schoonship is completely general and will expand \((a+b)^{30}\) to \(2^{30}\) terms sorting them afterwards. The reason for this is rather simple: if \(a\) and \(b\) are noncommuting objects there would indeed be \(2^{30}\) different terms! So we work our way around this by defining a binomial expansion:

$$X \quad \text{bin}(n^-, a^-, b^-) = a^n \cdot DS [J, 0, n, ((b/a)^J)JB, ((n+1-JB)/JB)]$$

The \(X\) is a different use of the substitution. It defines a substitution that should be attempted at all levels of the computation. It is an automatic substitution and therefore very suited to construct ‘hidden’ built-in knowledge. Its disadvantage is that because it is tried at all times it is rather time consuming. Therefore a responsible way of building rules makes a mixed use of \(X\) statements (which could also be hidden inside one block that is read in at the begin of the program) and blocks that are applied only whenever they are needed.

The \(n^-, a^-, b^-\) indicate wildcarding. As \(n^-\) indicates the quantified variable \(n\) it is clear to the program that the occurrence of \(n\) on the right-hand side of the formula corresponds to it.

We are now in the position to build the real program (table 2).
We see here how the expansions of the \( \sin \) and the \( \log \) are invoked successively, as the first call to expand only acts on \( \sin \) or \( \log \) with the argument \( x_3 \). When \( z^{n-} \) is substituted this involves a bin function (which is declared to be automatic in the \( x \) statement) and it is replaced immediately. Next \( xp \) are \( yp \) are substituted in one pass. To understand this (and the meaning of the \( Al \) statement) one has to scan a term for a match. Using the \( Al \) statement causes it to look simultaneously for the substitution in the \( Al \) statement after it looked for the one from the \( Id \) but before it makes the replacements. This is very useful sometimes. It is the responsibility of the user to avoid nasty interferences of the two substitutions. If undesired effects are possible one should use 2 \( Id \) statements, rather than one \( Id \) and one \( Al \). In the above program the \( xp \) and \( yp \) are completely unrelated so there is no problem.

The only three unknowns left are now

1. the function declarations
   
   The \( F \) statement declares \( \sin \) and \( \log \)s to be functions.

2. The \( A \ x = 31 \) declaration
   
   This means that there will be an automatic rule that \( x^{\mathfrak{31}} = 0 \) so the series expansion does not suffer from non-relevant partial results for higher powers of \( x \).

3. The numeric instruction \( N \)
   
   The computation of the binomial coefficients and the addition of the various terms causes the internal rationalization procedure to see some of the coefficients as irrational. The \( R10 \) tells it that if it only considers the first 10 (decimal) digits everything is fine after which the rationalization is executed properly.
Running the above program on an Atari ST gives after about 10 seconds:

```plaintext
> BLOCK expan(x,J2)
> 1d log1(x)=DS[J,1,'J2',('x^J'),(-1/(J-1)/J)]
> 1d sin(x)=DS[J,1,'J2'/2,('x^2'),(-1/(2*J-1)/(J-1/2))]*x
> ENDBLOCK
    A  a,b,y,p,x=31,sin(x,logf,z
    X  bin(n,-a,-b)=a^n*DS[J,0,n,(b/a)^J],[((n+1-J)/J)]
    F  log1,sin
    Z  f=a*sin(x)+b*logf/x^5
L1  1d sin(x)=sin(y)
L1  Al logf=log1(z)
>  expan(y,30)
L2  1d log1(y)=DS[J,1,30,('y^J'),(-1/(J-1/J)]
L5  1d sin(y)=DS[J,1,30/2,('y^2'),(-1/(2^J-1)/(J-1/2))]*y
> expan(z,35)
L8  1d log1(z)=DS[J,1,35,('z^J'),(-1/(J-1/J)]
L11 1d sin(z)=DS[J,1,35/2,('z^2'),(-1/(2^J-1)/(J-1/2))]*z
L14 1d y=x^3
L15 1d z^n=bin(n,x,y)
L19 1d xp=x
L19  Al y=x^2
N R10
> P stat
    "end"
```

\[ f = + a^x^3 - 1/6^a^x^9 + 1/120^a^x^15 - 1/5040^a^x^21 + 1/362880^a^x^27 
- 1/5^b^x - 1/3^b^x^3 - 1/7^b^x^2 + 1/8^b^x^3 + 2/9^b^x^4 + 1/10^b^x^5 
- 1/11^b^x^6 - 1/6^b^x^7 - 1/13^b^x^8 + 1/14^b^x^9 + 2/15^b^x^{10} 
+ 1/16^b^x^{11} - 1/17^b^x^{12} - 1/9^b^x^{13} - 1/19^b^x^{14} 
+ 1/20^b^x^{15} + 2/21^b^x^{16} + 1/22^b^x^{17} - 1/23^b^x^{18} 
- 1/12^b^x^{19} - 1/25^b^x^{20} + 1/26^b^x^{21} + 2/27^b^x^{22} 
- 1/30^b^x^{23} + 1/29^b^x^{24} - 1/15^b^x^{25} + 1/31^b^x^{26} 
+ 1/32^b^x^{27} + 2/33^b^x^{28} + 1/34^b^x^{29} + 1/35^b^x^{30} 
- b^x^4 + 1/2^b^x^3 - 2/3^b^x^2 + 1/4^b^x^1 + 0. \]

**(III) Derivation of the Campbell-Baker-Hausdorff formula**

In this chapter we will see an example of interplay between Schoonschip and the user that will lead to rather complicated results after a few steps.

Suppose one works with noncommuting objects like matrices, elements of a Lie algebra etc. One can define for such an object \( A \):

\[ e^A = 1 + A + A^2/2! + A^3/3! + ... \]

The C-B-H formula for two of those objects \( A \) and \( B \) is:
\[ e^{A+B} = e^A e^B e^{-\frac{1}{2} [A,B]} e^{-\frac{1}{2} [A,[A,B]]/6} e^{-\frac{1}{2} [[A,B],B]/3} e^{-i \text{ constant}}. \] (4 objects).

In normal quantum mechanics one works with \( A \sim p \sim i \hbar \partial / \partial x \), \( B \sim x \) and \( i \hbar \partial / \partial x \sim x \). All higher commutators are zero and we see just the phase factor \( e^{-i \text{ constant}} \). In a more general case one might be interested in obtaining more terms in this expansion. We will use the following strategy:

1. Evaluate \( e^{A+B} \) to a given 'precision',
2. Multiply from the left with \( e^{-A} \),
3. Multiply from the left with \( e^{-B} \),
4. Keep multiplying with known terms till we are finished with all terms to 'precision - 1'. Then the terms in 'precision' are the first terms of the Taylor expansion of some exponents.
5. Use these exponents now as input to the next job to obtain the next higher precision.

The execution of these steps is rather simple:

\[
\begin{align*}
F & \quad A,B \\
X & \quad \text{COMMU} (A,B) = A*B - B*A \\
X & \quad \text{EXP}(A-J) = \text{DS}[J,0J,(A\cdot J),(1/J)] \\
F & \quad b,x,y,c,c2 \\
A & \quad \text{power} = 3 \\
Z & \quad f = \text{EXP}(b,2) \\
Id & \quad b = (x+y) \cdot \text{power} \\
Keep & \quad f \\
\text{*next} & \quad f1 = \text{EXP}(c,2)\cdot\text{EXP}(b,2)\cdot f \\
Id & \quad b = -x \cdot \text{power} \\
AI & \quad c = -y \cdot \text{power} \\
Id & \quad \text{power} = 1 \\
\text{*end} & \\
\end{align*}
\]

The variable power counts the combined powers of \( x \) and \( y \) and all terms in power^2 and higher powers are eliminated due to the declaration \( A \cdot \text{power}=3 \). The answer gives therefore the second order terms:

\[ f1 = 1 - \frac{1}{2} x y + \frac{1}{2} y x. \]

These are of course the zeroth and first order terms in the expansion of \( e^{-i \hbar [x,y]} \). So in the next job we rerun the first job with \( A \cdot \text{powers} = 4 \), all second coefficients of the EXP calls should be 3 to get the third order terms and the definition of \( f1 \) should now be

91
\[ Z \quad f_1 = \text{EXP}(c_2,2) \cdot \text{EXP}(c,3) \cdot \text{EXP}(b,3) \cdot f \]

with

\[ \text{ID} \quad c_2 = \text{power}^2 \cdot \text{COMM} (x,y) / 2. \]

This gives the cubic terms. This process can be repeated several times although it must be remarked that the comptutetime increases rather rapidly when the higher and higher terms are considered.

**IV) Solving equations**

Suppose one needs to solve two equations in \( x \) and \( y \):

\[
\begin{align*}
    f_1 &= a_1x + b_1y + c_1 = 0, \\
    f_2 &= a_2x + b_2y + c_2 = 0.
\end{align*}
\]

Using Macsyma one would just give it the instruction to solve these equations and the answer would soon come on the screen. How does one do this on Schoonschip or how would one proceed with Macsyma if the equations are too difficult for a general approach?

The standard way is to define

\[ f_3 = a_1'f_1 - a_2f_2 = 0. \]

This would give

\[ f_3 = (a_1' b - ab')y + a_1' c - ac' = 0 \]

and the solution becomes trivial.

This is a completely general method that can also be used for nonlinear equations:

\[
\begin{align*}
    f_1 &= a_1 x^2 + b_1 x + c_1 = 0, \\
    f_2 &= a_2 x^2 + b_2 x + c_2 = 0.
\end{align*}
\]

with \( a_i, b_i, c_i \) functions of \( y \).

One would define:

\[
\begin{align*}
    f_3 &= a_2 f_1 - a_1 f_2 \quad (=0) \\
          &= (a_2 b_1 - a_1 b_2)x + (a_2 c_1 - a_1 c_2)
\end{align*}
\]
and next:

\[ f_4 = a_1 x f_3 - (a_2 b_1 - a_1 b_2) f_1 = 0 \]
\[ = \text{linear in } x . \]

Finally, \( f_5 \) combines the two linear equations \( f_3 \) and \( f_4 \) to get one equation that is independent of \( x \). The point here is that there is usually much freedom in choosing which terms to eliminate first and which equations are best for the later stage. A program that has a single general algorithm might fail where repeated running to build up simpler and simpler equations (in number of variables, not necessarily in numbers of terms) could lead to a solution. A good example from ref.2 is the set of coupled equations

\[
\begin{align*}
    x_1(x_2 + x_3) - 2a &= 0, \\
    x_2(x_3 + x_1) - 2b &= 0, \\
    x_3(x_1 + x_2) - 2c &= 0.
\end{align*}
\]

The general algorithms of Macsyma take about 1 1/2 minutes to solve this. Using the above method enables a person to do this by hand in hardly more time. This means that it is best to leave the formula crunching to the computer while the user does the thinking if the problem gets really big.

It shows again the difference between the consumer class and the research class approaches: the average Macsyma user will be happy to wait this long as he does not have to worry. The average Schoonschip user will think that such a time to solve such a simple problem is ridiculous.

A little messy example of linear equations is given below:

Suppose one needs to know the total resistance of a tetrahedron with 6 (different) resistors as its edges:
The linear equations are according to Ohm’s law:

\[ F_1 = V_A \cdot V_C - I_1 \cdot R_1 \]
\[ F_2 = V_A \cdot V_E - I_2 \cdot R_2 \]
\[ F_3 = V_C \cdot V_B - I_3 \cdot R_3 \]
\[ F_4 = V_E - V_B - I_4 \cdot R_4 \]
\[ F_5 = V_A \cdot V_B - I_5 \cdot R_5 \]
\[ F_6 = V_C \cdot V_E - I_6 \cdot R_6 \]

Substitute the current conservation by

\[ I_5 = I_{TOT} - I_1 - I_2 \]
\[ I_4 = I_2 + I_6 \]
\[ I_3 = I_1 - I_6 \]

Next one keeps making runs of the program. After each program one subtracts some multiples of the equations from each other to eliminate variables (this may need a good editor) and runs the next job till the final equation becomes after

\[ I_5 = V_A = dV - VB \]

as in figure 2:

C
C Program to find the resistance of a tetrad with 6 different resistors as its sides.

C r1, r2, r3, r4, r5, r6
A va, vb, vc, ve, dv
A i1, i2, i3, i4, i5, i6, itot
Z f1 = va - vc - i1 \cdot r1
Z f2 = va - ve - i2 \cdot r2
Z f3 = vc - vb - i3 \cdot r3
Z f4 = ve - vb - i4 \cdot r4
Z f5 = va - vb - i5 \cdot r5
Z f6 = vc - ve - i6 \cdot r6
L 1 Id i5 = itot - i1 - i2
L 1 Al i4 = i2 + i6
L 1 Al i3 = i1 - i6
Keep f1, i2, i3, i4, i5, i6

> P noutput
  *next
  Z h1 = f1 + f3
  Z h2 = f2 + f4
  Z h3 = f5
  Z h4 = f1 + f6 + f4
L 2 Id va = vb + dv
B i1, i2, i6
Keep h1, h2, h3, h4

> P noutput
  *next

94
\[ Z \quad k_1 = r_4 h_4 + (r_4 + r_6) h_1 \\
Z \quad k_2 = r_3 h_2 + r_4 h_1 \\
Z \quad k_3 = h_3 \\
B \quad i_1, i_2, i_6 \\
\text{Keep} \quad k_1, k_2, k_3 \\
\text{> P output} \\
\text{"next"} \\
Z \quad m_1 = r_4 k_2 - (r_2 + r_4) k_1 \\
Z \quad m_2 = r_5 k_1 + r_3 r_4 k_3 \\
B \quad i_1, i_2 \\
\text{Keep} \quad m_1, m_2 \\
\text{> P output} \\
\text{"next"} \\
Z \quad n_1 = (m_1 * ( - r_1 * r_3 * r_5 - r_1 * r_4 * r_5 - r_1 * r_5 ^ 2 - r_3 * r_5 ^ 2 ) \\
\text{-} m_2 * ( r_1 * r_2 * r_3 + r_1 * r_2 * r_4 + r_1 * r_2 * r_6 + r_1 * r_3 * r_4 + r_1 * r_4 * r_6 \\
+ r_2 * r_3 * r_4 + r_2 * r_3 * r_6 + r_3 * r_4 * r_6 ) ) / r_3 / r_4 \\
B \quad d\nu, \nu_t \nu \\
\text{> P output} \\
\text{"end"} \\

n_1 = \\
+ d\nu \\
\text{*( } - r_1 * r_2 * r_3 - r_1 * r_2 * r_4 - r_1 * r_2 * r_6 - r_1 * r_3 * r_4 - r_1 * r_3 * r_5 \\
- r_1 * r_4 * r_5 - r_1 * r_4 * r_6 - r_1 * r_5 * r_6 - r_2 * r_3 * r_4 - r_2 * r_3 * r_5 - r_2 * r_3 * r_6 \\
- r_2 * r_4 * r_5 - r_2 * r_5 * r_6 - r_3 * r_4 * r_6 - r_3 * r_5 * r_6 - r_4 * r_5 * r_6 ) \\
+ \text{ itot } \\
\text{*( } r_1 * r_2 * r_3 * r_5 + r_1 * r_2 * r_4 * r_5 + r_1 * r_2 * r_5 * r_6 + r_1 * r_3 * r_4 * r_5 \\
+ r_1 * r_4 * r_5 * r_6 + r_2 * r_3 * r_4 * r_5 + r_2 * r_3 * r_5 * r_6 + r_3 * r_4 * r_5 * r_6 ) + 0.

The final answer is now:

\[
R_{\text{tot}} = \frac{\text{coef of } \text{itot}}{\text{coef of } d\nu}
\]

(V) An example from High Energy Physics

In this chapter we will look at a "simple" reaction that caused some problems to many graduate students about a decade ago. The reaction is \( e^+ e^- \rightarrow \tau^+ \tau^- \rightarrow \nu_\tau du \) ud \( \nu_\tau \) as given in figure 3.
Using the Feynman rules it is rather easy to write down a highly condensed formula for the transition matrix element of this reaction:

\[
|M|^2 = \begin{vmatrix}
  u(p_n)\gamma_\mu(1+\gamma_5)(p_\mu+im_\mu)\gamma_\nu(\gamma_\nu+im_\nu)\gamma_\nu(1+\gamma_5)v(q_n) \\
  u(p_d)\gamma_\mu(1+\gamma_5)v(q_u) \\
  u(p_d)\gamma_\nu(1+\gamma_5)v(q_d) \\
  v(q_e)\gamma_\nu u(p_e)
\end{vmatrix}^2.
\]

The Schoonschip program for this problem can be rather short but suffers a shipwreck after 2500 seconds while generating more than \(2 \cdot 10^5\) terms occupying 8 Mbytes of disk space (there being no more of that).

This forces the user to some tricks which are rather straightforward to run the program of figure 4.

\[
\begin{align*}
A & \text{ mtau, melec, mup, mdown} \\
V & \text{ pe, qe, q, pt, qt, pn, qn, pu, qu, pd, qd} \\
I & \text{ mu, nu, ka, al, be, ga} \\
Z & \text{ ampa(nu) = } G(1, nu) * G(6,1) * Ug(1,0, qn) \\
Z & \text{ ampb(mu) = } Ubg(1,0, pn) * G(1, mu) * G(6, 1) \\
Z & \text{ ampc(ku) = } (G(1, pt) + i^* \text{ mtau}) * G(1, ka)^* * (G(1, qt) + i^* \text{ mtau}) \\
Z & \text{ amp2(nu) = } Ubg(2, mup, pu) * G(2, nu) * G(6, 2) * Ug(2, - mdown, qd) \\
Z & \text{ amp3(mu) = } Ubg(3, mdown, pd) * G(3, mu) * G(6, 3) * Ug(3, - mup, qu) \\
Z & \text{ amp4(ku) = } Ubg(4, - melec, qe) * G(4, ka) * Ug(4, melec, pe) \\
\text{ Keep} & \text{ ampa, ampb, ampc, amp2, amp3, amp4} \\
\text{ next} & \\
\text{ ampa(nu) = } + G(1, nu) * G(6, 1) * Ug(1,0, qn) \\
\text{ ampb(mu) = } + Ubg(1,0, pn) * G(1, mu) * G(6, 1) \\
\text{ ampc(ku) = } \\
& + G(1, ka) \\
& * ( - \text{ mtau} * 2 ) \\
& + G(1, ka) * G(1, qt) \\
& * ( - i^* \text{ mtau} ) \\
& + G(1, pt) * G(1, ka) \\
& * ( i^* \text{ mtau} ) \\
& - G(1, pt) * G(1, ka) * G(1, qt) \\
\text{ amp2(nu) = } + Ubg(2, mup, pu) * G(2, nu) * G(6, 2) * Ug(2, - mdown, qd) \\
\text{ amp3(mu) = } + Ubg(3, mdown, pd) * G(3, mu) * G(6, 3) * Ug(3, - mup, qu) \\
\text{ amp4(ku) = } + Ubg(4, - melec, qe) * G(4, ka) * Ug(4, melec, pe) + 0.
\end{align*}
\]

\[
\text{ Sum} \quad \text{ nu, be, mu, al} \\
Z & \text{ mat2 = ampa(nu) * Conjug(ampa(be))} \\
& * \text{ amp2(nu) * Conjug(amp2(be))} \\
Z & \text{ mat3 = Conjug(ampb(al)) * ampb(mu)} \\
& * \text{ amp3(mu) * Conjug(amp3(al))} \\
Z & \text{ mat4 = amp4(ku) * Conjug(amp4(ga))}
\]
L 8  Id,Spin,1,2,3,4
L 9  Id,Trick,Trace,2,3,4
L14  Id,Trick,1
L19  Id,peDqe=qDq/2+melec**2
L20  Id,qe(ka)=pe(ka)
L20  Al,qe(ga)=pe(ga)
    Keep  ampc,mat2,mat3,mat4
    *next

    mat2 =
    + G(1,pu)
    * (64*i*qnDqd )

    + G5(1)*G(1,pu)
    * (-64*i*qnDqd )

    mat3 =
    + G(1,qu)
    * (64*i*pnDpd )

    + G5(1)*G(1,qu)
    * (-64*i*pnDpd )

    mat4 = -8*pe(ka)*pe(ga)

    + D(ka,ga)
    * (-2*qDq ) + 0.

    Z  mat=mat3*ampc(ka)*mat2*Conjg(ampc(ga))*mat4

L 4  Id,Trick,Trace,1
L 9  Id  ptDpt=-mtau**2
L 9  Al  qtDq=-mtau**2
L 9  Al  ptDq=qDq/2+mtau**2
    *yep
L 1  Id  peDqe=qDq/2+melec**2
L 2  Id,Dotpr,pe(ka-)=q(ka)-pe(ka)
L 3  Id  qDpt=qDq/2
L 3  Al  qDq=qDq/2
    B  pnDpd,qnDqd
    *end

    mat =
    + pnDpd*qnDqd
    * (-262144*mtau*2*peDpe*qDq*puDqu + 524288*mtau*2*peDpe*ptDpu*ptDqu
    + 524288*mtau*2*peDpe*ptDpu*qtDqu + 524288*mtau*2*peDpe*ptDqu*qtDpu
    + 524288*mtau*2*peDpe*qtDpu*qtDqu + 1048576*mtau*2*peDpt*peDqt*puDqu
    - 1048576*mtau*2*peDpt*peDpu*ptDqu - 1048576*mtau*2*peDpt*peDpu
    *qtDqu - 1048576*mtau*2*peDpt*peDqu*ptDpu - 1048576*mtau*2*peDpt
    *peDpu*qtDpu + 524288*mtau*2*peDpu*peDqu*qtDqu + 262144*mtau*2*qDq
    *ptDpu*ptDqu + 262144*mtau*2*qDq*qtDpu*qtDqu + 262144*mtau*4*qDq
    *puDqu + 524288*peDpe*qDq*ptDqu*qtDpu - 2097152*peDpt*peDqu*ptDqu
    *qtDpu + 262144*qDq*2*ptDqu*qtDpu ) + 0.

We see again some declarations of algebraic symbols (A), vectors (V) and indices (I). The expression of M is split in six pieces which are each typed in separately, and kept for further processing. In the next step, rather than combining them all immediately they are put together partially in $|M|^2$. The instructions that
are involved form a rare example of a Schoonschip knowledge base. It has some special instructions for the operations that are called for here:

Conjug is the conjugation operator. It reverses the order of functions and applies complex conjugations to the built-in complex functions Ug, UbG, G and the imaginary variable i. The summation over the external spins: Id, Spin

Taking traces of strings of γ matrices G. Id, Trick, Trace

The rearrangement of γ matrices in 4 dimensions Id, Trick

These operations are very efficient unless one tries to use many of them simultaneously in the same expression. This is why the job in which one tries to get the whole amplitude in one step fails so horribly. In the current job they are done separately so that the answer can be simplified before the next step, which is putting it all together.

Quantities as peDq are fourvector products $p_e q_e$. The final step puts it all together, takes the long trace and makes some very obvious simplifications. The total runtime of this job is between 2 and 6 seconds (the clock has a 2 seconds resolution).

(VI) Building a knowledge base

Here we will look at the construction of a simple library to be used for differentiation of a restricted class of functions. To do this properly we have to consider the "product rule":

$$\frac{d}{dx} (f(x)g(x)) = \frac{df(x)}{dx} g(x) + f(x) \frac{dg(x)}{dx}.$$  

How does this work out for products of $x^3$, sin, cos and log? Consider the block:

```
BLOCK differ {x,ddx}
Id 'ddx'*x'^n~ = n'*x'^n(n-1) + 'ddx'*x'^n
Id 'ddx'*sin(x') = cos(x') + 'ddx'*sin(x')
Id 'ddx'*cos(x') = -sin(x') + 'ddx'*cos(x')
Id 'ddx'*log(x') = x'^-1 + 'ddx'*log(x')
Id 'ddx' = 0
```

The block will apply the chain rule by 'differentiating' the various functions separately. After each step one of the functions has had its turn and is not considered any further. The terms with the derivatives have no more ddx so there cannot be double work. Finally there is no untreated function left so the remaining term is removed by putting ddx equal to zero.

Testing this explicitly:

$$f = x^3\sin(x)\cdot ddx$$

$$\Rightarrow 3x^2\sin(x) + x^3\sin(x)\cdot ddx$$

$$\Rightarrow 3x^2\sin(x) + x^3\cos(x) + x^3\sin(x)\cdot ddx$$

$$\Rightarrow 3x^2\sin(x) + x^3\cos(x).$$

This is indeed the correct answer.
The next complication is of course the chain rule:

\[
\frac{d(f(g(x)))}{dx} = \frac{dg(x)}{dx} \cdot \left( \frac{df}{dx} \right)(g(x)).
\]

This is solved with statements of the type

\[
\text{Id} \quad \ddx*\sin(z) = \diff(x,z)\cos(z) + \ddx*\sin(z).
\]

Afterwards one has to specify what \(\diff(x,z)\) means via

\[
\text{Id} \quad \diff(x,x) = 1
\]
or

\[
\text{Id} \quad \diff(x,x) = \cos(x)
\]

if

\[z = \sin(x).\]

These schemes can be extended further and further depending on what is needed and how ingenious one can be.

Integration is somewhat more complicated. One way is to make a big table with all relevant integrals in it. This involves statements of the type

\[
\text{Id} \quad dx^x^n = x^{n+1}/(n+1).
\]

There must be a strict ordering in such a table to avoid calamities when products of functions are considered. The presence of the \(dx\) avoids again a double treatment. These tables have however a tendency of becoming very large and the use of them will be very time-consuming.

Therefore there is much work on algorithms to to partial integrations to bring the integrals to a standard form, or even on methods to construct the integral. The greatest benefits lie till now in the field of formulae with many not too difficult integrals. Here the results are very impressive. If difficult integrals are involved one will still have to do them by hand.

A very impressive knowledge base was constructed by J.J. van der Bij and M. Veltman for two loop computations in the standard SU(2)×U(1) model. They put all Feynman rules of the model in a 'formula base' and generated with Schoonschip thousands of two loop diagrams. This resulted in a giant two loop correction computation of several physics parameters \((p, M_W, M_Z)\).
BUSES FOR HIGH ENERGY PHYSICS

W. von Rüden

CERN, Geneva, Switzerland

Abstract

This lecture tries to answer some of the questions concerning buses for data acquisition brought up by today's high-energy physics experiments. It recalls the principal limitations of CAMAC before discussing the characteristics of FASTBUS and VME in some detail as they are the preferred systems for new experiments. G64 is mentioned, being used for detector survey and control. The new VAXBI bus as successor to UNIBUS gets some attention, because VAX computers seem to have become a de facto standard. Most experiments use a combination of the above systems and they all have to be connected together. Interfacing to buses becomes an issue and local area networks assume a growing importance in distributed systems; Ethernet and UETNet fulfill some of these needs. The ALEPH and UA1 experiments are used to give examples, the choice being determined by the knowledge of the author rather than any other criterion. A general introduction to buses can be found in the proceedings of a previous CERN School.

1. THE PROBLEM IN HIGH-ENERGY PHYSICS

Today's high-energy physics experiments show an ever increasing number of detector channels with more and more complex electronics. Half a million channels and raw data rates of up to 500 Megabytes per second are not unusual any more. It is obvious, that such systems have to use better readout systems than CAMAC, developed at a time where an experiment would fit on the top of a table and was done by some ten physicists. Today we talk about 400 people in one collaboration and experiments that look like large factories, with the difference, that the data rates are one hundred times higher.

The large number of channels requires a large board size at least at the level of the front-end electronics; the use of ECL circuits, FADCs and fast memories leads to power supply capacities of up to 3 kW per crate, which means up to 10kW per rack, with the consequence that cooling becomes a non-trivial task. The quoted data rates cannot be recorded on magnetic tape and on-line data reduction is found in all experiments. Imbedded microprocessors such as the M68020 with a computing power comparable to mini-computers are found in large numbers, special purpose trigger processors help to decide on events to be taken or rejected and computing engines such as emulators reduce the data even further before they enter the data acquisition computer.

In addition to the complexity of such an experiment, frequent changes of running conditions are expected. Some part of a detector may be ready while another part is being calibrated, yet another one being tuned or built up. Such activities must be possible simultaneously and can be achieved by a proper portioning of the readout system.

Last, but not least, a large number of institutes contribute to the construction of the detector and hence the use of standards at least at the level of the experiment is vital to achieve
compatibility between the different components. The equipment will have to be maintained over many years to come and usually improvement programmes are being discussed at about the same time as the detector delivers its first data. Flexibility is the keyword to solve this problem (at least in theory) and 'low-cost' is the other well beloved expression; unfortunately they are usually incompatible.

2. **WHY NOT CAMAC**

CAMAC has now been used for many years successfully for many experiments, but the larger experiments had to use special CAMAC implementations such as REMUS to overcome the limitation imposed by the standard. The board size is not large enough, nor are the power supplies, and the mechanics are rather expensive. The real limitations come from the structure of the system being conceived originally as a single master system and meant to be an extension to the I/O system of the mini-computer. The later introduction of the GEC Elliot system crate philosophy improved the situation substantially, but could still not overcome the limitations imposed by the addressing scheme. A maximum of 49 crates distributed over 7 branches is possible per system crate. The internal module addressing is only 4 bits wide, only 5 bits are reserved for internal functions and there is no direct access from crate to crate. The data path is 24 bits wide (no longer enough for 32 bit microprocessors and mini-computers) and the readout speed is limited to 0.5 to 2 Mbyte/s for block transfers.

The existing investment in CAMAC is very high and many functions do not yet exist in either VME or FASTBUS. Therefore, CAMAC is expected to coexist, in the form of sub-systems, with more modern bus systems.

3. **FASTBUS**

FASTBUS has been conceived as a modular multi-master, multi-crate data acquisition and control system with the aim to overcome the deficiencies encountered in CAMAC and has been accepted as an ANSI/IEEE standard 2). It would exceed the scope of this lecture to give a complete tutorial on FASTBUS and we refer to other publications 3-4) for some introductory information. We will review the main characteristics and concentrate more on the application aspects.

3.1 **Basic FASTBUS elements**

Before entering any discussion at a more technical level, we introduce the basic elements found in any FASTBUS system and their symbolic representation (see Table 1). The most basic element is called a *segment* defined as an autonomous bus interconnecting *master* and *slave* devices. Segments appear as crate segments and cable segments with almost identical properties. The bus protocols governing the communication between master and slave devices are the same for both types of segment and therefore modules are addressed in the same way on a cable as in a crate. The symbol for the crate segment includes the terminators, whereas for cable segments the terminators are shown explicitly. The *ancillary logic* is not shown, but it is a vital element and must be present on any segment to make it work. It generates timing signals and is involved in the the arbitration mechanism. It either plugs into the back of the
crate segment or it exists for the cable segment as a FASTBUS module powered from the crate.

**TABLE 1**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Master</td>
</tr>
<tr>
<td>S</td>
<td>Slave</td>
</tr>
<tr>
<td></td>
<td>Crate Segment</td>
</tr>
<tr>
<td></td>
<td>Cable Segment</td>
</tr>
<tr>
<td>T</td>
<td>Terminator</td>
</tr>
<tr>
<td>SI</td>
<td>Segment Interconnect</td>
</tr>
<tr>
<td>PI</td>
<td>Processor Interface</td>
</tr>
</tbody>
</table>

*Segment Interconnect* modules provide the connection between cable and crate segments and they contain routing tables necessary to establish the path between the source master and the destination slave, which may reside several segments away.

Fig. 1 2) shows a simple three crate system with a host computer and its processor interface connected to a cable segment. Most PI implementations found today consist of a FASTBUS part residing in a crate segment with a special cable going to the computer part sitting on the computers I/O bus.

![FASTBUS System Topology Diagram](image)

*Fig.1 Example of FASTBUS System Topology*
FASTBUS allows many computers to be connected to the system but only one is defined to be the host, i.e. the master which initializes the whole system.

A FASTBUS crate without cooling and power supply is sketched in Fig. 2. Only the lower backplane is mandatory as it carries the power supply lines and all the signals for the crate segment. The upper auxiliary backplane exists with 2 or 3-row connectors with 130 or 195 pins respectively and is not bused. It is used either to connect private signals between modules or in the case of the SI or other units on a cable segment, to carry the cable segment signals.

![Fig. 2 Layout of a FASTBUS crate](image)

3.2 FASTBUS signals

Table 2 is a summary of FASTBUS signals. The function of most signals is evident from the name and only some remarks are made here for less obvious signals. The precise definition and the function of each signal as well as the timing are defined by the standard 2).

The pair AS/AK is used to establish a lock between a master and a slave and this lock can only be released by the master, even if a higher priority device request the bus. Therefore a master can block part or all of a FASTBUS system if for example a processor crashes while the lock is established. In such a case RB is usually the only way out. Data transfers are initiated by the master with DS and acknowledged by the slave through DK. In block transfer mode data is exchanged at each transition of DK (as opposed to most other bus systems, where transfers only occur on one edge).

BH is generated by the ancilliary logic in response to a signal from a switch on the front bar of the crate. This bar blocks the modules mechanically and has to be lowered before a
module can be inserted or removed. This feature should allow modules to be changed without
switching off the power supply, the BH signal preventing the other modules in the crate from
reacting to spurious signals.

The T pin on each slot is wired on the backplane to the address line corresponding to the
slot’s geographical address, i.e. the T pin of slot 5 is connected to AD05. It serves to identify
modules having issued a service request SR or having data to be read in sparse data scan.

**TABLE 2**

**FASTBUS Signals**

<table>
<thead>
<tr>
<th>RB</th>
<th>Reset bus</th>
<th>AK</th>
<th>Address acknowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>Address sync.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EG</td>
<td>Enable geographical</td>
<td>AD (32)</td>
<td>Address / data lines</td>
</tr>
<tr>
<td>MS (3)</td>
<td>Mode select</td>
<td>SS (3)</td>
<td>Slave status</td>
</tr>
<tr>
<td>DS</td>
<td>Data sync.</td>
<td>DK</td>
<td>Data acknowledge</td>
</tr>
<tr>
<td>RD</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE</td>
<td>Parity enable</td>
<td>PA</td>
<td>Parity</td>
</tr>
<tr>
<td>AL (6)</td>
<td>Arbitration level</td>
<td>AI</td>
<td>Arbitration inhibit</td>
</tr>
<tr>
<td>AR</td>
<td>Arbitration request</td>
<td>AG</td>
<td>Arbitration grant</td>
</tr>
<tr>
<td>GK</td>
<td>Grant acknowledge</td>
<td>SR</td>
<td>Service request</td>
</tr>
<tr>
<td>WT</td>
<td>Wait</td>
<td>BH</td>
<td>Bus halted</td>
</tr>
<tr>
<td>GA (5)</td>
<td>Geographical address</td>
<td>TP</td>
<td>T pin</td>
</tr>
<tr>
<td>DL, DR</td>
<td>Daisy chain left, daisy chain right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TX, RX</td>
<td>Serial Network lines</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3 **FASTBUS addressing scheme**

FASTBUS provides three addressing modes: *logical, geographical and broadcast.*

Geographical addressing is always needed, the two other modes are optional. On a crate
segment each slot has 5 encoded pins (GA) defining the geographical address from 0 to 25
from the right to the left; on cable segments the address is selected by switches on the module.

Upon system startup, the host will use this addressing mode to initialize all modules in the
system, in particular the SIs to establish the possible connections between the segments. As
part of this procedure some modules may be set to respond to logical addresses. This mode can
make the application programs independent of the physical location of a module in the system.

Broadcast addressing is used to send messages to several modules at a time or to do a sparse
data scan via T-pin reads. In addition these modes can be applied to either data space or
control/status register (CSR) space both with a 32-bit address range. Table 3 summarizes the
different addressing modes.
TABLE 3
FASTBUS Addressing Modes

<table>
<thead>
<tr>
<th>EG</th>
<th>MS&lt;2:0&gt;</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Logical in data space</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Logical in CSR space</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Broadcast to data space</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>Broadcast to CSR space</td>
</tr>
<tr>
<td>0</td>
<td>4-7</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Geographical in data space</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Geographical in CSR space</td>
</tr>
<tr>
<td>1</td>
<td>2-7</td>
<td>Illegal</td>
</tr>
</tbody>
</table>

The basic transaction for a handshake read operation for geographical or logical addressing is as follows. The master asserts the primary address (module address) on AD followed by AS, the selected slave answers with AK. The master then asserts the read line RD and the data sync DS, the slave will place the data on AD and answer with DK. The master stores the data, releases RD and DS, the slave in turn releasing DK. If no further data cycle is needed, the master will release AS, and the slave, AK, terminating the transaction. This is a simplified description, not dealing with bus arbitration, parity or error codes. The standard document should be consulted for a complete description. Eight different types of data cycle are possible, five being defined and three being reserved. In the most simple case, secondary address (MS=2) or random data (MS=0) cycles will be used.

All FASTBUS addresses are 32 bits wide to address modules in the same way on the segment where the master resides as on any other segment in the system. The addressing formats for the three addressing modes are as shown in Fig. 3. Unfortunately, some of the boundaries are implementation dependent and could be different from system to system. In practice the group field is eight bits, a choice introduced through the implementation of the Segment Interconnect.

<table>
<thead>
<tr>
<th>Logical:</th>
</tr>
</thead>
<tbody>
<tr>
<td>31  ?  ?  0</td>
</tr>
<tr>
<td>Group Module Internal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geographical:</th>
</tr>
</thead>
<tbody>
<tr>
<td>31  7  0</td>
</tr>
<tr>
<td>Group 0 GA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Broadcast:</th>
</tr>
</thead>
<tbody>
<tr>
<td>31  7  0</td>
</tr>
<tr>
<td>Group X· Condition</td>
</tr>
</tbody>
</table>

Fig. 3 Addressing Formats
3.4 FASTBUS arbitration

The arbitration in FASTBUS is fully symmetrical, either on the level of a segment or system wide. No special slot or arbiter unit is needed. It would go too far to treat this complex matter here, but the basic principle is interesting enough to be mentioned. Forgetting about signal delays and cable length etc. it works as follows: the master asserts an arbitration request AR and when arbitration grant AG is received from the ATC logic it places its arbitration level on AL.<05:00> of the bus. It then compares its own value with the value on the bus (the logical OR of all competing masters) and if the internal value of a given bit differs from the value on the corresponding AL line of the bus, all bits of lower significance are removed by the master. If the remaining priority on the bus matches the master's priority level, it concludes that it has won the arbitration and asserts grant acknowledge GK. We illustrate a simple case with three masters in the following example.

Example:

<table>
<thead>
<tr>
<th>Level</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>20</td>
<td>7</td>
<td></td>
<td>31</td>
<td>30</td>
<td>26</td>
</tr>
</tbody>
</table>

| AL5  | 0  | 0  | 0  | 0  | 0  | 0  |
| AL4  | 1  | 1  | 0  | 1  | 1  | 1  |
| AL3  | 1  | 0  | 0  | 1  | 1  | 1  |
| AL2  | 0  | 1  | 1  | 1  | 1  | 0  |
| AL1  | 1  | 0  | 1  | 1  | 1  | 1  |
| AL0  | 0  | 0  | 1  | 1  | 0  | 0  |

The masters M1, M2 and M3 have priority level 26, 20 and 7 resp. and we look at the levels found on the AL lines at times T1 to T3. At T1 all bits are present (OR) and let's say M3 sees that bit 3 of his level differs from AL3. It removes all lower bits (2-0) and we find the state at T2. Now M2 finds the same situation for bit 3 and therefore removes bit 2. What remains on the AL lines at T3 is the level of M1 which recognises the match and decides that it is the winner. Arbitration levels 1-31 are used within a segment and must be uniquely assigned, levels 32-63 are system levels and must be unique along a given route.

Depending upon the way a master starts its arbitration request, the standard defines three modes which are Assured Access, Prioritized Access and Direct.

4. THE USE OF FASTBUS IN ALEPH

The Aleph collaboration has decided to use FASTBUS for data acquisition. Being aware of the very complex nature of this new standard, it was decided to keep the system as simple as possible, yet allowing for flexible use of the equipment, in particular to partition it dynamically.
The experiment is composed of several sub-detectors with a total of about 140 FASTBUS crates. The general philosophy is to build a hierarchical system with a tree structure having intelligent modules being masters towards the front-end and slaves towards the on-line computers. In this way it is possible to skip readout stages by connecting the next higher master to a slave; this is not possible in systems with dual master or dual slave units. This hierarchical structure excludes by definition communications between devices on the same level which is possible with the hardware. Only geographical and broadcast addressing are used.

4.1 Sub-detector readout

Fig. 4 shows a typical configuration for one sub-detector. The front-end electronics is read by a readout controller (ROC) being master on the crate segment and slave on the cable segment. Each branch connects via an SI to the central crate where the Event Builder (EB) is the controlling element. The EB has two cable segment slave ports going to the sub-detector computer and the central readout respectively. For normal data acquisition the ROCs will read out and format in parallel the data in their crates and then inform the EB via service request (SR) that they have data. The EB does a broadcast T pin read to determine the source(s) of the SR and proceeds to the readout after which it will pass on the data to the main Event Builder and/or to the sub-detector computer.

Fig. 4 Readout of a sub-detector
The link between the trigger system and the ROCs is via dedicated cables carrying signals such as the bunch crossing from the LEP machine, trigger level 1 yes or no, trigger level 2 yes or no, readout controller busy, etc. Each ROC is connected via a 16 channel bidirectional link using complementary ECL signals to a Fan In Fan Out (FIO) box serving as a programmable switching node between the ROCs and the local or central Trigger Supervisor. This setup allows the attachment of an ROC to either of the two input ports (main or local), or to disconnect it completely, providing the necessary hardware to support the partitioning of the system to the level of an ROC. Partitioning is used to bring up the system step by step, to allow independent activities like calibration or debugging or to exclude faulty crates temporarily from the readout. More details about this system are found in a report of the Aleph Dataflow Group 5).

The FASTBUS standard foresees only a single service request line per segment for the slave to call the attention of a master. Systems with multiple masters dealing with multiple slaves on the same segment are therefore forced to use a 'service request handler' which will receive all service requests, translate them into interrupt messages and transmit them to the master taking care of a particular slave. As it is very difficult to build a general purpose service request handler purely in hardware, we profit from the power of the Event Builder to incorporate this function as an independent task.

One of the general features of the Aleph data acquisition system not shown in Fig. 4 is the Cheapernet local area network connecting all ROCs and EBs to the on-line computers. The main purpose of this connection is to have a second path to the embedded microcomputers, mainly 68020s, for diagnostics in case FASTBUS hangs up, to readout histograms generated locally and to login remotely for debugging. Even low speed data acquisition is not excluded.

4.2 Central readout

Going higher up in the system we see in Fig. 5 how these sub-detectors are connected to the VAX Cluster. Larger sub-detectors have their own VAX and therefore the Event Builder has two output ports, one for the sub-detector VAX and the other for the central readout. Depending on the mode of operation one of the ports may be used to 'spy' on the data passing through the second port. The EB of small sub-detectors has only one output port.

The central readout and the handling of the small sub-detectors is done in the main VAX host, which also controls additional CPUs called 'Event Processor' for on-line event filtering. We expect a total of 4-6 machines in the final configuration. The interfacing to the VAX is planned to be done with the future CERN Host Interface (CHI) connected via a fibre optics link to an I/O port on the VAXBI bus. Due to the long distance between the FASTBUS crates and the computers, pipelined data transfer will be needed to achieve the required data rates.

The initialisation of such a FASTBUS system is a very complicated task, if partitioning and resource management need to be incorporated. A joint project between Aleph, DD division and Delphi is taking care of this by providing a general purpose software tool adaptable to different FASTBUS systems. It is based on the entity relationship model and uses a database to store module and configuration descriptions.
4.3 The Aleph Event Builder

The operating systems of most minicomputers introduce a rather high overhead for I/O operations if a driver is needed to allow for multi user access to the data acquisition system. The solution to this problem is either to incorporate list processing inside the driver or to move the readout task out to intelligent front-end machines. If multiple tasks need to be executed in these front-ends the problem returns, unless the machine itself efficiently supports multi-user operations.
The Aleph Event Builder has been conceived with these considerations in mind and it is worth spending a chapter on it because it employs a novel technique to interface a 32 bit microcomputer to FASTBUS. Complete FASTBUS transactions are handled by a FASTBUS coprocessor of the M68020 CPU directly rather than via calls to a library. This coprocessor adds new instructions to the standard M68020 instruction set, in the same way as the floating point coprocessor, making FASTBUS a native bus of this enhanced CPU. All FASTBUS operations are executed as single instructions (Fig. 6 shows a simplified programmers model). For example, a 'single word read from data space' instruction is FBSWRD A_i, A_j, D_k, where A_i and A_j have to be loaded with the primary and secondary addresses and D_k will return the data value. In the same way as a standard M68020, reference to non-existing memory would generate a bus error trap. FASTBUS errors will also generate traps, avoiding error checking after each FASTBUS instruction. Of course, one needs to write a trap handler either as part of the operating system or, for more sophisticated applications, as a user trap handler.

In the block diagram of the unit in Fig. 7 the M68020 and the Lance chip for Cheaperernet are able to access both the processor bus and the memory bus. This dual bus structure has been chosen to allow normal processor operations and network activities while the FASTBUS coprocessor transfers data from the master port on the crate segment to one or both event memories. Access to the memory bus is controlled via an arbitration circuit allowing interleaved access from the CPU and the FASTBUS coprocessor. The event memories are dual ported and accessible from the cable segment slave port. In the most complicated case one could have four concurrent activities taking place on the same memory, one event being read in from the master port, the previous one being treated by the CPU and the one before being read from the cable segment, while a histogram is being transferred via the Local area network.
The coprocessor contains a microcoded engine which takes care of the communication with the M68020 and the very complex FASTBUS protocol, including arbitration, timeouts, error handling, service requests and interrupt messages. To support this hardware an extended assembler and a Fortran compiler generating in-line code for FASTBUS calls are used. Other languages can be adapted via user written functions or macros. It is planned to revise the FASTBUS calls to conform to the new standard routines being finalised now.

5. THE VME BUS

VME stands for VersaModules Europe and was proposed by Motorola, Mostek, Signetics/Philips and Hitachi. It is derived from Versamodules, Motorola's proprietary bus system for the Exormacs computer, the first machine based on their M68000 chip. Driven by the strong interest of the European industry in products built on boards with the 'Eurocard' form factor the multi company venture had a good chance to become an industry standard and in deed, today more then 100 manufacturers all over the world support it.

5.1 General VME Features

VME has been conceived as a single crate system not a general purpose data acquisition system. Fig. 8 shows a crate having a terminated backplane with up to 20 slots with TTL tristate signals; the protocol allows data rates of up to 20 Mbytes per second. The boards are double height Eurocards with two 96-pin DIN connectors. The bus is asynchronous and non multiplexed and comes in two flavours, one using only the upper connector J1 with 24 address lines and 16 data lines, the second using in addition the centre row of the lower connector J2 with 8 extra address lines and 16 more data lines. The outer rows of J2 are for private I/O signals or for various extension buses.
Fig. 8 Sketch of a 20 slot VME crate and a VME board

One such bus is Motorola's I/O bus based on single height Eurocard modules, an approach not followed by most of the VME producers and hence not very attractive.

VMX has been designed as a memory extension bus with the aim to provide fast access to private additional memory which cannot, for space reasons, fit on the CPU board. It uses the two outer rows of connector J2 and due to the limited number of pins (64) it...
multiplexes addresses and data. Often this extension is used for a direct I/O channel into the CPU but as there are no interrupt lines on VMX, the use is somewhat limited. Unfortunately, there are already 3 revisions of the specifications and the early users of VMX have compatibility problems now. Anyhow, the whole VMX philosophy is being superseded by the new VSB proposal which found more support within the VME users community.

![Diagram of VME/VMX/VMS Bus Architecture](image)

**Fig. 9 VME/VMX/VMS Bus Architecture**

VMS is a *serial communication bus* implemented with two lines called serial clock SERCLK and serial data SERDAT on the VME backplane, without a defined protocol.

Boards in slot #1 of each crate have to provide the function of the system controller, supplying the 16 MHz system clock to the bus, taking care of arbitration, system reset, etc.

### 5.2 VME Addressing

The VME bus has been tailored to fit the M68000 family CPU signals, adding some general purpose signals. Other CPUs can be interfaced to the bus as well. All boards are directly addressed in the CPUs address space and 6 additional lines called 'address modifiers' (AM0 ... AM5) are defined providing a kind of simple memory management. In practice these lines are not used very frequently. Out of the total 64 possibilities, 16 are user defined codes ($10 - $1F), many codes are reserved for future use. A few examples are

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3E</td>
<td>Standard Supervisory Program Access</td>
</tr>
<tr>
<td>$3A</td>
<td>Standard Non-privileged Program Access</td>
</tr>
<tr>
<td>$39</td>
<td>Standard Non-privileged Data Access</td>
</tr>
<tr>
<td>$2D</td>
<td>Short Supervisory Access</td>
</tr>
<tr>
<td>$29</td>
<td>Short Non-privileged Access</td>
</tr>
</tbody>
</table>

To simplify the decoding for input/output board an I/O page with 16 bit addressing ranging from $FF0000 to $FFFFFF has been defined with its own address modifier codes.
5.3 VME Arbitration

Unlike FASTBUS or VAXBI with symmetric arbitration VME needs special circuitry in slot #1 for the arbitration. Four levels of bus requests BR0-3 with the corresponding bus grant signals BG0-3 and a bus busy BBSY signal are defined with BR3 having the highest and BR0 the lowest priority. Within the same arbitration level the priority decreases from the left to the right within a crate, because the grant signal is daisy-chained. Empty slots in a crate must be equipped with jumpers in case any unit in a higher slot needs to become a bus master. Boards such as memories which do not use arbitration signals are supposed to link them through passively. Fig. 10 shows how these lines are wired on the backplane.

![Diagram of VME Arbitration Daisy Chain](image)

**Fig. 10 Arbitration Daisy Chain in VME**

5.4 VME Interrupts

The VME system provides seven interrupt levels and seven interrupt lines IRQ1 to IRQ7 on the backplane. As with the arbitration the interrupt acknowledge is daisy chained, but only one line is used, the priority level being given by a code on address lines A01-A03. Three types of functional modules are used with interrupts: Interrupters, Interrupt Handlers and Daisy Chain Drivers. The interrupter pulls a certain interrupt line down, let's say IRQ4, the interrupt handler in charge of this level will try to become bus master and then execute an interrupt acknowledge cycle by placing the priority code on the address lines and by driving IACK low. This bus line is connected to IACKIN on slot #1 from where it propagates through IACKOUT to IACKIN on the next slot and so on. Any slot not involved in the sequence just passes the signal on (don't forget the jumper in empty slots) and the slot with the lowest slot number having requested an interrupt at the particular level will respond with Status/ID information to be used by the interrupt handler. VME also allows multiple interrupt handlers in which case the protocol gets more complicated. Refer to the VME documentation.
6. **VME IN UA1**

The UA1 experiment was the first one at CERN to employ VME in a large style for data acquisition. The large number of readout channels and the increasing need for on-line computing together with the severe limitations of CAMAC where the driving forces to move to a new technology. Due to the parallel nature of the readout system, the combination of VME and VMX was the appropriate solution. Fig. 11 shows an overview of the UA1 readout system.

![Diagram of the UA1 VME Readout System](image)

*Fig. 11 The UA1 VME Readout System*
The basic processing node consists of a CPUA1 processor with a dual-ported VMX/VME memory module and a readout controller (driver). The CPU and the I/O driver are in the upper VME crate, the dual ported memory is located in the lower VME crate and the connection is made by extending the VMX bus vertically between the two crates. This solution works very well but does not conform to the VMX definition which allows only a cable long enough to cover 5 slots within the same crate. UA1 started in the early days of VME and therefore revision A of VMX is still used. This dual crate system connects via a **Crate Interconnect Bus** to a second dual crate system arranged symmetrically to the first one. While in the first system each node collects a piece of the same event from a source in parallel with other nodes, a node in the second system receives a full event and the different nodes execute different task on the data. The I/O channels serve now for output and drive, for example, a bank of 168E or 3081E emulators or generate event displays. A new readout system can be added to the upper part by adding another node with its specific I/O driver as long as there is space in the crate, and a new function can be added to the lower part independently.

Going through the diagram from the top to the bottom we distinguish nine different system buses with distinct functions implemented on VME, VMX or dedicated connections. First there is the **detector bus** bringing in the data from the front end electronics to a nodes memory. These nodes are supervised via the **readout control bus** in VME. The next is a **local bus** in VMX connecting the readout driver and the CPU in a node to its dual-ported memory, of which the second port resides on the **event data bus**. On this bus events are transfered via a **crate interconnect bus** to a second **event data bus** with a different functionality. Again a **local bus** connects the dual-ported memories in the processing nodes to the CPU and the output driver. The processing nodes are controlled via the **peripheral bus** again in VME. The last ones are the **output buses** actually driving the peripherals.

Many of the VME crates are controlled via a modified MacIntosh personal computer called **MacVEE**, which can access the VME address space directly as its own memory. This solution combines the flexibility of VME with the user-friendliness of the MacIntosh. Up to eight VME crates or seven VME crates plus seven CAMAC crates may be connected to one MacVEE via flat cables with a total length of up to 100 m.

7. **THE G64 BUS**

G64 is a small bus system based on single height Eurocards; it was developed at CERN and in industry and it is supported today by some 10 companies in different European countries. Originally designed for the M6809 8-bit microprocessor it now supports several CPUs including the M68000 and even the M68020.

The basic version of G64 has 16 address and 8 data lines and uses a synchronous 1 MHz clock. The low system cost and the availability of the FLEX and OS-9 operating systems with high-level languages make it an attractive solution for accelerator control and **Slow Control** in large experiments, covering alarms, power supply survey, interlocks, gas control, etc. A wide range of standard modules such as ADCs, DACs, digital I/O, drivers for stepping motors and network devices are available from industry. In most cases the development system and the application program use the same crate.
What is missing in G64 is a generalised way to adapt the signals and connectors of the standard modules to the signals and connectors from the experimental equipment. The Aleph on-line group has developed a Monitoring and Control crate MAC64 based on G64 to overcome these limitations. This crate has a 15 slot G64 backplane in the lower part of the crate and a second special backplane with 12 slots above it. Slots 13 to 15 on the G64 bus are reserved for the processor board and the UTInet local area network adapter. A memory board or a floppy disk controller may be added if needed. Slots 1 to 12 are for application modules, each one needing an adapter board in the slot above it. The adapter board is used for signal conditioning like opto coupling, thermocouple adapting or amplification. The connection to the external world goes via 64 contacts on the back (which are not bused) of the upper backplane. The space between the upper backplane and the back panel is used to adapt the wiring of the top board to the connectors appropriate for external connections. The top backplane connectors have a third row of 32 contacts used as an identification bus. Each adapter board has its unique model number which can be read from the G64 bus through a connection on the back. This feature is used to automatically configure the system at power up. The processor reads the identification of the adapter in a given slot, knows therefore which card is underneath and how to initialise it. It then reports its configuration via UTInet to the central computers to be checked against the information in a database.

Each chassis has its own computer readable serial number which is used to assign its network address. In this way, all CPU boards run the same software and can be exchanged between crates. All crates are connected to UTInet, a CERN developed low-cost local area network, which is interfaced via a gateway to Ethernet for control from the VAX computers.

8. VAXBI

VAXBI or VAX Backplane Interconnect is a new bus system from Digital Equipment Corporation replacing the Unibus for the new VAX 8xxx family of computers (with the exception of the 86xx series). Small systems (VAX8200/8300) use BI as a system bus for CPUs, memory and I/O, while BI is used as an I/O bus in mid-range (VAX 8500/8550) and large VAX systems (VAX8700/8800). BI is intended to overcome the problems found with Unibus, such as interface implementations by third parties using their own design to connect to the backplane or addressing problems due to wrong selection of base addresses leading to double addressing. Fault isolation was mainly done by removing suspicious boards, a tedious job for boards with a lot of cables hanging on.

Fig. 12 shows the layout of a BI board with the BI corner implementing a unique way of connecting the backplane to any board. A VLSI chip called BIIC incorporates all the necessary functions in such a way, that BI boards do not have any switches or jumpers, all parameters being programmable. Base addresses and block size of memory boards are defined by writing into the BIIC rather than setting jumpers. Therefore, a CPU can reconfigure its memory in case a memory board is faulty at boot time. The BIIC also incorporates a self-test feature and the possibility of being disconnected from the bus under software control.
Another very nice feature is the use of zero insertion force connectors, which solves a well-known problem with bus connectors: either it is easy to insert and to remove the boards, in which case the force per contact is low and the contact poor, or the contacts are good and it becomes almost impossible to get the boards in and out. The BI uses 120 contacts for the bus itself and 180 contacts for private connections to other boards or for input/output. No other connectors are allowed on the board. The layout of the BI corner has been fully simulated to guarantee predictable behaviour under varying configurations.

Card cages come with 6 slots and can go up to 24 slots with a total length not exceeding 1.5 meters. For the initial and unique addressing of the control registers in the BIIC, the system uses keys on the back of the backplane, carrying the hexadecimal numbers 0 to F, which means that the maximum number of nodes, i.e. BI addressable boards, is limited to 16. BI nodes may consist of several boards and occupy more than one slot, but only one board is then equipped with the BI corner.

The BI bus has been conceived as a multi-processor bus (including protocols for distributed cache support) and uses synchronous protocols with imbedded arbitration. The sustained bandwidth is up to 13.3 Mbytes/s when used with more than one master and about 11 Mbytes/s for single masters. This difference is due to the imbedded arbitration cycle in which the current bus master is not involved, i.e. in a single master configuration there is an unused cycle in each transaction.
The transactions defined by the BI protocol are

- Command/address cycle
- Imbedded arbitration cycle
- Distributed arbitration (3 modes)
- Data cycles
  1 per Byte, Word or Longword
  2 per Quadword
  4 per Octaword
- Cache support for multi processors
- Interlocked operations for queues
- Multi-responder device interrupts
- Inter-processor Interrupts

The various processor configurations available today are explained in the following. The first example in Fig. 13 shows a small system with one processor, memory and I/O all on the BI bus. This configuration corresponds to a VAX8200 in a stand-alone version. The CPU power is equivalent to one VAX11/780, but the I/O bandwidth is higher.

The second example (Fig. 14) shows a VAX8300 which has two processors of the same type as in the 8200, but otherwise is the same. For the time being the VMS operating system supports only an asymmetric operation similar to the way the VAX11/782 is operated, but there are plans to make it symmetrical. From the number of nodes or slots it would be possible to add another CPU to the bus, but as all memory references go via BI, a three processor system would be limited by the bandwidth of the bus. The situation is different for processing nodes with private memory where only I/O and communication goes via the main bus.
The next higher CPU configuration corresponding to a VAX8500 is sketched in Fig. 15. The processor and the memory are no longer on BI, but on a separate high-speed memory bus. The BI is now only used for I/O and it connects to the memory bus via a VAXBI adapter using one node, leaving space for 15 more nodes on BI.

In the high-end VAX versions such as the VAX8700 with one processor or the VAX8800 with two processors (Fig. 16) the CPUs share a separate high-speed memory bus and the BI bus is used for I/O subsystems. Up to four BIs with 15 nodes each (one is used by the adapter) can be connected. Such a capacity will probably never be used for I/O, but becomes interesting if additional processors can be connected. In fact, BI allows for such extensions, even if today such a product has not yet been announced (Fig. 17).

The new VAXBI bus opens many interesting possibilities, but it is not yet clear what Digitals policy will be concerning the licensing of the BI technology. Manufacturers of DEC compatible equipment are of course very interested to have access to it, but as all products must use the BIIC VLSI chip designed and produced by DEC, the company can easily control the market. Up to now BI has not been licensed outside the United States.
Fig. 16 Multiprocessor High-End System Configuration

Fig. 17 High-End System Configuration with Additional Processors
9. CONCLUSIONS

For several years now, when a new experiment plans the data acquisition system, the question "FASTBUS or VME?" comes up and if the answer was simple the collaborations would not spend weeks or months in animated discussions to come to a decision. We try to summarise the main characteristics without pretending to have the answer.

9.1 FASTBUS

The advantages are: Conceived as a multi-master, multi-crate data acquisition system with generous addressing capabilities, large board size, good cooling, unified crate segment and cable segment approach, a good arbitration scheme, flexible broadcast system and recently standardised software.

The disadvantages are: Very complex protocols with too many options introducing significant overheads in hardware and software, rather high cost, very limited industrial support, very slow startup of production even for such basic items as crates and segment interconnects, high power consumption due to the use of ECL logic in many units. There is only one service request line making multi-master system implementations difficult. Backplane connectors are poor.

9.2 VME

The advantages are: Proposed and supported by many major companies, good mechanics with solid connectors, reasonable cost due to strong competition, easy to handle, large variety of processors, memories, I/O interfaces, complete systems available. Software support from industry with operating systems such as Versados, OS-9, CP/M and various real-time kernels.

The disadvantages are: Very small board size leading to 'piggy back' implementations and layouts with small chips sitting below large chips; incompatible with the M68020's dynamic bus sizing protocols; VMX32 is a 'fiddle' (VSB should do a better job). Up to now there is no multi-crate standard defined (do it yourself). Off-the-shelf modules are usually insufficient for High Energy Physics applications and the UA1 example shows that most of the modules had to be specified by the experimenters: however industry is very interested to take these developments up, which is not the case for FASTBUS developments (with a few exceptions). Despite much simpler bus transactions than those in FASTBUS, there are compatibility problems when using boards from different suppliers within the same crate, in particular in multi master applications.

9.3 The choice of the LEP experiments

It is not surprising that the four LEP experiments did not chose the same solution. Aleph, Delphi and L3 are mostly FASTBUS orientated with some VME equipment, Opal uses FASTBUS only for some front-end electronics and employs VME in the readout system and for data reduction. G64 is used by Aleph and Delphi, L3 and Opal prefer VME to monitor the detector. Common to all four experiments is the choice of VAX computers with BI or Qbus and Ethernet, yet with different software and network protocols.
REFERENCES


2) The Institute of Electrical and Electronics Engineers (IEEE), IEEE Standard FASTBUS Modular High-Speed Data Acquisition and Control System, ANSI/IEEE Std 960-1986


5) Aleph Dataflow Group, Aleph Data Acquisition System Hardware Functional Specifications, DATACQ note 85-21, CERN

6) VME System Architecture Manual


9) S. Cittolin, private communication
THE UA1 VME DATA ACQUISITION SYSTEM

S. Cittolin
CERN, Geneva, Switzerland

Abstract

The data acquisition system of a large-scale experiment such as UA1, running at the CERN proton-antiproton collider, has to cope with very high data rates and to perform sophisticated triggering and filtering in order to select and analyze interesting events. These functions are performed by a variety of programmable units organized in a parallel multi-processor system whose central architecture is based on the industry-standard VME/VMXbus. These lecture notes give a brief overview of the different stages and trigger levels of the UA1 data acquisition system and describe the general structure of the VME system hardware and software.

1. Introduction

The UA1 experiment is a large multi-purpose particle physics detector system designed to provide full solid-angle coverage around the LSS5 intersection region of the CERN proton-antiproton collider. The experiment started data-taking in November 1981, and during the last five years more than $10^{12}$ collisions have been observed and $10^7$ events have been analyzed. An improvement program is at present being implemented, involving the construction of a new calorimeter and more sophisticated trigger processors, to allow the experiment to run with the higher luminosity that will be delivered by the SPS collider when it is upgraded by the Antiproton Collector (ACOL) from the end of 1987.

![Diagram of the UA1 detector](image)

Figure 1. The UA1 detector
The various parts of the detector have already been described elsewhere [1]. Briefly, the apparatus (see Fig. 1) consists of a central detector built of drift chambers with image readout to provide particle tracking, surrounded by a variety of complementary electromagnetic and hadronic calorimeters in both the transverse and longitudinal directions of the beam.

Table 1

<table>
<thead>
<tr>
<th>Detector</th>
<th>No. channels</th>
<th>Raw data</th>
<th>Formatted data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central detector</td>
<td>6200</td>
<td>1600000 Bytes</td>
<td>≈80000 Bytes</td>
</tr>
<tr>
<td>Hadron calorimeter</td>
<td>1200</td>
<td>2400</td>
<td>2400</td>
</tr>
<tr>
<td>Electromagnetic cal.</td>
<td>2200</td>
<td>4400</td>
<td>4400</td>
</tr>
<tr>
<td>Cal. position detector</td>
<td>4000</td>
<td>8000</td>
<td>8000</td>
</tr>
<tr>
<td>Forward chamber</td>
<td>2000</td>
<td>32000</td>
<td>8000</td>
</tr>
<tr>
<td>Muon chamber</td>
<td>6000</td>
<td>≈1000</td>
<td>1000</td>
</tr>
<tr>
<td>Streamer tubes</td>
<td>50000</td>
<td>50000</td>
<td>≈4000</td>
</tr>
<tr>
<td><em>Uranium calorimeter</em></td>
<td>20000</td>
<td>80000</td>
<td>8000 (ACOL upgrade)</td>
</tr>
</tbody>
</table>

Collision rate (with a luminosity = $10^{30}$ cm$^{-2}$ s$^{-1}$) 50000 Hz
Average event size 160 Kbytes
Parallel readout and data reduction dead time 3-40 ms
Maximum event rate at third level trigger input 30 Hz
Maximum event rate recorded to mass storage 4 Hz

A dipole magnetic field provides momentum analysis, and the whole detector is surrounded by an iron shield instrumented with multiple planes of streamer tubes and a matrix of muon chambers. The data digitization and formatting are performed by electronic modules in about 200 CAMAC crates grouped in 28 REMUS branches. These are read in parallel into a multi-event buffer system. A variety of specialized processors, such as ROP, M68010/20, 168E, 3081E, SuperCAVIAR and NORD computers, share the tasks of event data compression, digital trigger, readout control and mass storage. The characteristics of these units have been described elsewhere [2].

The numbers of channels and the amount of data generated by the digitizing electronics for each event are given in Table 1 and the main components of the UA1 data acquisition system are listed in Table 2.

2. Trigger levels and data acquisition phases

At the design luminosity of the SPS proton-antiproton collider ($10^{30}$ cm$^{-2}$ sec$^{-1}$) the collision rate is of the order of 50 kHz, with a beam crossing every 3.8 μsec. The average luminosity obtained during the last data-taking period, in 1985, was $3\times10^{29}$ cm$^{-2}$ sec$^{-1}$ corresponding to a collision rate of about 15 kHz. However, the maximum event readout rate was limited by the transfer rate of the mass storage (magnetic tape or video disk with a maximum throughput of 0.5 Mbyte/sec) to about 4-5 Hz. In order to reduce the trigger frequency from several kHz to a few Hz, and maintain the overall system efficiency above 90%, the data acquisition operates in separate stages, each of which communicates with the next by means of multi-event buffer systems (see Fig. 2).
Table 2
Data acquisition components

<table>
<thead>
<tr>
<th>VME modules</th>
<th>70</th>
<th>Dual port memory VME/VMX DPRX 128/256Kb Static RAM.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>VME crate interconnect.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>256Kb EPROM.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1Mb DPDX Dual Port memory VME/VMX Dynamic RAM.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Parallel I/O, interrupt generator, graphics ....</td>
</tr>
<tr>
<td>Readout</td>
<td>200</td>
<td>Data digitization</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>Parallel readout and event builder</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>VME crates.</td>
</tr>
<tr>
<td>8 bit Processors</td>
<td>200</td>
<td>Electronics control</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>Data reduction/formatting</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Equipment test and control</td>
</tr>
<tr>
<td>16 bit Processors</td>
<td>7</td>
<td>FAMP M68000.</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>Parallel readout, data formatting and event data sampling</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Robcon VME20 M68020 1Mb, M68881 FP.</td>
</tr>
<tr>
<td>32 bit Processors</td>
<td>6</td>
<td>168E IBM 1Mbyte.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3081E IBM 3.5Mbyte.</td>
</tr>
<tr>
<td>Main Computers</td>
<td>2</td>
<td>NORD 100/500 2 Mbyte.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6250 bpi 125 ips.</td>
</tr>
</tbody>
</table>

The data acquisition system trigger is organized in a number of separate levels, some of which operate concurrently with the processes of data digitization, data reduction, and data readout.

Beam-beam interactions are selected by a pre-trigger using standard NIM logic which demands a coincidence between hodoscopes in the proton and antiproton directions to within ±20 ns. A first-level decision is then made between beam crossings, using a purpose-built processor system to fast identify energy distributions and prompt muon signals from the muon chambers. During this time data are either in the phase of digitization (central detector, muon chamber) or in hold state (calorimeter signals, streamer tubes, etc.). If the event is accepted the digitization is completed and the data reduction and reformattting phases can be enabled.

A second-level trigger decision, using M68000 microprocessors to attempt to establish whether the muon candidate comes from the interaction region, is activated only if a muon candidate is identified at the first level. In order to further refine the selection with a more detailed description of the energy distribution in the calorimeters, this level will be improved with M68020 and hardwired processors which find each cluster and measure its energy. It is expected that the event rate will be reduced to 30 Hz after this level.

If a trigger is generated, the data stored in the front-end buffers are read and formatted in parallel by a set of specialized processors and saved into a multi-event buffer system.

The system dead-time is determined mainly by the central detector data-processing time (of the order of 40ms). However, a hardware double buffer input to each digitizer channel of the central detector readout reduces this dead-time further to the ADC conversion time of 3ms. In the upgrade of the experiment this feature will be extended to all the detector front-end electronics, reducing the dead time of this stage to about 200μs.
Finally more refined decisions are made, based on the analysis of the event data by a stack of 168E and 3081E emulators. These processors re-check the trigger condition matched in the preceding levels with greater precision and reconstruct part of the event using the information produced by all the detectors. This event filter is used either to flag or to reject an event. The maximum input rate to this phase of the readout is about 30 Hz while the output rate, which is limited by the mass storage speed, must be reduced to 4 Hz.
3. The data acquisition

The main purpose of a data acquisition system is to read the data from the experiment's instrumentation and to store to tape that corresponding to physically interesting events. This must be done with the maximum of efficiency and while providing a continuous monitor of the data validity and the detector performance.

The high trigger rate and the large data volume of the detector information demand a powerful and versatile data acquisition system. The complexity of the equipment requires continuous monitoring with programmable means of error detection and error recovery. The data have to be reformatted and compacted in size and the event rate has to be controlled and reduced by programmable filters. So a large amount of computing power has to be made available as close to the detector as possible, a high degree of parallelism has to be implemented and sequential processes have to be pipelined as much as possible to limit the dead time and to maximize the effective use of the experiment run time.

In order to realise all these features and performance, the data acquisition must be founded on a flexible and powerful system with distributed multi-processing power. The UA1 data readout is based on the VME/VMX/VSBbus standards, using M68K family microprocessors to control the data flow, MacVEE systems for program development, control and monitoring, and CERN/SLAC 168E and 3081E IBM emulators to analyze the event data.

The VMEbus is a 32-bit data and 32-bit address asynchronous multi-processor bus [3] introduced by industry (Mostek, Motorola, Signetics, Thomson) in 1982. The VMXbus and VSBbus are compatible local busses. The bus mechanics is based on double-height Eurocards in 20-slot crates.

3.1 The logical structure

The logical structure of the readout (Fig. 3), consists of two main sub-systems, implementing the parallel readout and the event builder tasks. The first task is to read, reformat and buffer the front-end data while the second one is to assemble the data blocks from the individual subsections of the detector into a full event record and to distribute it to a set of event processing units. These tasks are executed by two processors called the readout supervisor and the event manager.

The parallel readout task is executed by a set of data input units, the parallel readout units, each designed to deal autonomously with the associated detector front-end digitizing electronics. In a similar way the event builder task is executed by a set of event processing units, the event units. An event unit is a complete computer system (CPU, program and data memory, input/output link) dedicated to process a full event data record according to an assigned sub-task (event filter, event display, mass storage etc.).

The parallel readout and the event builder tasks operate independently, the first responding to the readout trigger and the second servicing all the event requests coming from any of the event units. When an event is processed by the primary event unit (the unit dedicated to the mass storage) the event buffer space used in the parallel readout is freed for a following event.

The data flow between the two main tasks is derandomized by a buffer with multi-event storage capability and operating in the first-in/first-out event mode. A common event directory memory supplies the control information to handle the input event FIFO and the output event requests queue. A data path, connecting the two sub-systems, provides the fast event data block transfer. The logical buses connecting these basic units together can be grouped into two classes, the global and local buses:
3.2 Global buses

A global bus is a general-purpose bus with a multi-processor arbitration protocol hosting master and slave devices. A master device is a unit able to access the bus and generate a bus cycle, while a slave device is a unit able to recognize a bus cycle and to respond to it. The global bus is implemented physically by a set of VME crates with a multi-crate interconnection.

Fig. 4 shows the following global buses:

- Readout control bus. This bus hosts all the readout units and the readout supervisor processor. The bus is used by the readout supervisor to control all the phases of each detector data input. The readout supervisor has to initialize the readout units, to monitor the data flow and to check for timeout and errors. The processor running the task of readout supervisor controls the readout control bus.

- Event data bus. This bus has access to all the memories used in the data acquisition system. The bus is used to collect all the event data blocks from each readout unit, to assemble and store them, as a full event record, into the memory buffer of the requesting event units. The processor running the task of event manager controls the event data bus.

- Peripheral control bus. This is a public bus connecting a set of event units. It allows all the event unit processors to share a set of common resources such as graphics display, data links and mass storage. No main controller processor is assigned to this bus. Physically the bus is implemented by a single VME crate.
3.3 Local buses

A local bus is any connection among a fixed number of defined devices such as a front-end electronics board, a CAMAC crate controller, an input/output device, a processor or a memory module. The use of such a bus involves only the ports connected to it. Physically it is implemented by a cable segment. Fig. 5 shows the following local buses:

- **Detector bus.** The detector bus links the digitizing electronics modules to a controller unit, called the bus driver. A variety of bus detector implementations is in use at UA1, each adapted to particular front-end electronics systems. For example, the CERN REMUS bus is used to read all the equipment in CAMAC crates. A synchronous 8-bit TTL bus reads the streamer tube ADC electronics (STAR). An asynchronous 16-bit TTL bus reads and controls a LeCroy Fastbus processor [7], and 300 optical fibers link the new calorimeter digitizers to the VME readout modules. In each case the bus is chosen to suit the design of the front-end electronics and its location in the experiment set-up.
- **Private bus.** The private bus is a memory access bus which is used to communicate between a master device and a dual-port memory. The master device can be either a DMA channel or a processor unit. These buses are implemented with the VMX.A and VSBbus standards.

- **Output bus.** Like the detector bus, the output bus is used to communicate with external systems (e.g. main computer, serial Cernet data link, parallel TTL-CAMAC).

4. The VME implementation

The logical structure described above is quite common to many data acquisition systems. For experiments not having a high trigger rate or a large event data volume, the complete readout can be implemented by a single mini/medium-size computer running a multi-tasking operating system. Each parallel readout or event unit task is executed by a program module running on the same machine under the scheduling of the operating system, while the data input is performed serially by a single channel.

The use of VME allows a cost-effective approach to the implementation of a modular and flexible 'hardware' multi-tasking operating system with a high throughput (several Mbyte/sec). The system tasks are implemented by physical processors and message exchange is accomplished by memory modules which are accessible from a global bus.

A block diagram of the UA1 data acquisition VME implementation [4] is shown in Fig. 6. In order to exploit the full bus bandwidth and to avoid the bus contention caused by frequent accesses by several master units, the three global buses are physically separated and implemented by VME crates connected together by a bus extension system. The readout control bus and the event data bus are under the overall control of the readout supervisor and the event manager respectively. These bus controllers are used to synchronize and to monitor the operations of all the processors on the bus.

Each readout unit is interfaced to the system by a detector bus driver module, with VME and VMX access, and a dual-port memory. The detector bus driver is located in a VME crate of the readout control bus and the dual-port memory in a VME crate of the event data bus. The two modules are connected together by a VMX segment. During data-taking, the detector driver module, initialized by the readout supervisor processor from its VME port, reads and stores the data into the dual-port memory from its VMX port. The size of the memory, usually 128Kbytes, is sufficient for the storage of the subsystem data for several events.

The event unit system has a symmetrical structure. A dual-port memory of sufficient size to contain a full event record (256Kbytes) is located on the event data bus, and a CPU and an output driver are located in a VME crate acting as the peripheral control bus.

The present UA1 system consists of 30 parallel readout units and 12 event units. Since all the unit tasks are executed by independent physical processors using their local VMX segments for data access, the throughput of the system does not depend on the number of units but only on the speed of the input channels and the transfer rate of the VME crate interconnection DMA (which is at present 6.5 Mbyte/sec).

Since the VME and VMXbus specifications do not define multi-crate systems, special extension modules have been developed, and the application of the local VMXbus has been extended beyond its original specifications.
4.1 The VMXbus

The VMX specification defines a local bus whose function is to connect a primary and a secondary master to one or more slave modules (memory or input/output devices). A maximum of 6 units may be housed in the same VME crate (Fig. 7). The VMXbus allows the local memory space of a processor unit to be enhanced, while reserving VMEbus accesses for common data and the exchange of control and status information.

In our application this concept has been extended to allow the connection of modules in different crates as well (Fig. 8). For instance a master module in one crate can access a dual-port memory in a second crate (the electrical characteristics of the modules in use at UA1 allow this kind of extension to up to three crates). This configuration offers a powerful data communication link between two VME crates which can be used to transfer messages and data between a public VME bus (e.g. crate 2 in Fig. 8) and a processor system housed in another crate (crate 1 in Fig. 8). All the readout and event units are configured in this way.
4.2 The VME crate interconnect

Another aspect not covered by the VMEbus specification is that of multi-crate networks. A crate interconnect system has therefore been developed to allow the implementation of the functionality required by the global buses used in the UA1 scheme.
The crate interconnect [5] comprises a VME master and slave module driving a 32-bit external bus (the vertical bus) used to link together up to 16 units. Among these units only one, which is located in the VME crate of a bus controller (readout supervisor or event manager), acts as the master of the vertical bus. Since the main functions performed by the bus controller are device control and high-speed data transfer, the crate interconnect was designed to operate in two modes (Fig. 9) under the control of the crate interconnect master module. One mode is for control access (window mode) and the other for fast data block transfer (DMA mode).

- **Window mode.** The window mode allows a processor unit to handle a multi-VMEbus crate system as a single logical bus. From the master module a window of 64Kbytes can be mapped into a slave crate memory at a programmable address. This mode allows transparent access to any memory of a remote crate from a processor unit in the same VME crate as the crate interconnect master.

- **DMA mode.** In the DMA mode a data block of any length can be transferred between two crates, under the control of the master interconnect module. The 32-bit (or 16-bit) data transfer (at a speed of 2.5Mtransfers/sec) has a three-phase pipeline: the data read from the source memory (a VME cycle in the source crate), the vertical bus data transmission and the data write into destination memory (a VME cycle in the destination crate).

The crate interconnect vertical bus is also interfaced to CAMAC by means of a dedicated module acting as CAMAC auxiliary crate controller. Both crate interconnect operating modes are accessible from CAMAC functions.

An upgraded version of the crate interconnect system is currently under development. It will allow multi-master arbitration and longer inter-crate distances (kilometers) by means of optical links. This new module is based on an M68020 controller with VME and VSB master capabilities, using high-speed serial electrical and optical signals (140MBaud) for the data link. Fig. 10 indicates the physical layout of the VME system and the crate interconnections.

**Figure 9. The VME Crate Interconnect**
4.3 The VME M68010 UA1 standard processor unit

The CPUA1 module is the basic processor unit used in all the VME readout subsystems. Its main features are: a M68010 8MHz processor, VME/VMX bus master, 256Kb dynamic memory, 8Kb static memory dual-ported CPU/VME, NS16081 6MHz floating-point processor, MK68901 peripheral controller (interrupt handler, timer, RS232 serial interface), and a variety of control and status registers [6]. Recently a more powerful processor unit, the Robcon VME20, has been introduced. This is a VME/VSB master, based on the 16 MHz M68020 with M68881 coprocessor and 1Mbyte CPU/VME dual-port memory.

In both cases the processor module has a CPU/VME dual-port memory (8Kb in the CPUA1 and 1Mb in the VME20) with a hardware feature which generates an internal interrupt when
its first word location is written from VME. This dual-port memory is the basic means of
data communication between processors and it is used to activate programs, to exchange
messages or to emulate device registers.

4.4 The VME/VMX dual-port memory

The VME/VMX dual-port memory is the second basic module of the UA1 system. It has
been implemented in two versions:

A static RAM dual-port VME/VMX memory (DPRX) is used both for the parallel readout
and the event builder units. Its main features are: 128Kb/256Kb static RAM VME/VMX
dual-port memory, 32-bit (16-bit) data, 400ns 32-bit word data transfer on both ports, VME
VME memory base address programmable via a VME register, and write broadcast mode
implemented via address modifier selection. The latter mode allows the loading of several
event builder units by a single DMA transfer, when all the corresponding memory units are
set to the same VME address.

A version of this unit with 1Mbyte dynamic RAM (DPDX) is used as a local memory
extension of an event builder CPU.

5. The parallel readout unit

The VME parallel readout unit is the basic element of the data input system. In general it
consists (Fig. 11) of a detector bus driver, an optional central processor unit CPU, and a 128
Kbyte VME/VMX dual-port memory. The detector bus driver and the CPU are connected
via the VME port to the readout control bus and operate under the control of the readout
supervisor processor. The VME port of the dual-port memory is connected to the event data
bus. The VMX port shares a VMX segment with the CPU and the detector bus driver that
act as VMX primary and secondary master respectively.

A readout unit may not have an associated CPU, in which case all the readout control is
performed by the readout supervisor processor. During data acquisition each event unit takes
care of the detector data readout and buffering. Data reformatting and/or second-level trigger
selection are executed at the end of readout by the CPU accessing data via its local VMXbus
segment. The CPU is also responsible for the monitoring, calibration and system test tasks.

A detector bus driver can be a single VME module, such as the VME REMUS branch driver
[8] used to read the CAMAC electronics, or it can have a substructure as in the case of the
larocci STAR readout, where an entire VME crate performs the function of the input driver.

Figure 11. The parallel readout unit
5.1 The streamer tube parallel readout unit

The readout of the streamer tube muon detector is an example of a complex readout unit (Fig. 12). The system consists of 50000 channels, digitized in groups of 32 by ADC boards located near the detector chambers [9]. The ADC boards are linked together in 18 independent loops using byte-parallel TTL signals. Each loop is driven by a VME module, called the STAR controller, which is a VMX slave. The readout is performed by 8 CPUA1 VME processors, each connected via its VMX port to one or more STAR modules and housed in a separate VME crate. Each CPU reads and compacts the data coming from its STAR drivers and stores the formatted data into the local CPU-VME dual-port memory (8Kbytes). At the end of the readout one of the processors, acting as the readout crate master, reads the eight data blocks from the VME CPU ports, and saves the assembled event into the readout buffer memory connected to the CPU VMX segment. Optionally the event data are also written into the local memory of another processor used for data monitoring and calibration.

The system is interfaced to the general VME readout by two VME/VMX dual-port memories connected by a VMX segment to the readout crate master CPU. One memory is located on the VME readout control bus and it emulates the registers of a detector bus driver. The

![Diagram of the streamer tube readout unit](image)

Figure 12. The streamer tube readout unit

137
second memory is located on the VME event data bus and it performs the function of the multi-event data buffer for the STAR detector.

6. The event unit

The event unit has to accomplish the task of full event data analysis. It consists of one or more dual-port memories accessible by the event manager from the event data bus and by a local CPU from a VMX segment (Fig. 13). Optionally it can be accessed by an output driver unit for data communication with external systems. Each dual-port memory has a size of 256Kb, sufficient to store a full event record. The CPU and the output driver VME ports reside in an independent VME crate used as peripheral control bus. The task of an event builder is either to process a full event data record for data sampling, calculating statistics, histogramming and display purposes, or for data communication with the third-level trigger and the mass storage. In general, a single task is associated with each event unit and several such units can be inserted into the system in modular fashion without affecting the overall speed performance.

![Diagram of event unit](image)

Figure 13. The event unit

6.1 The emulator farm event unit

The IBM emulators (168E and 3081E [10]) perform as the event filter of the UA1 experiment. An emulator is a computing machine with a limited set of instructions and a fast access memory, complemented by fast integer and floating-point hardware coprocessor units. The processor register set and the design of the instructions are as near to those of a mainframe computer as possible (in this case the IBM 370/168 and 3081). Hence, after compilation of a program (usually in Fortran) on an IBM computer, the machine code, with the exception of the input-output instructions, can easily be translated into the larger more primitive set of emulator instructions. The execution of this code by the emulator produces the same bit-to-bit result as the original IBM computer.
This procedure has the advantage that all the software development and debugging is done in a comfortable environment (the central computer), using high level compiler optimization techniques, and in addition it is a very cost-effective solution to the online (and offline) high-level computing requirements of the experiment. Six 16B8B emulators have been used as the event filter in the past experiment data-taking periods. They will be replaced by a new VME event unit with six 3081E emulators, each having a memory of 3.5Mbytes and a computing power equivalent to an IBM 370/168.

Since each emulator has to process a complete event at a time, the processors are configured in a linear expandable stack using VME buses for input data and output data (Fig. 14). The memory system of each processor can be doubly accessed from the two crates through a VME interface able to control the emulator central processing unit and to map the emulator memory into the VME address space. Two VME CPUA1 control the data traffic between the data acquisition input and the output mass storage device (NORD link and VME mass storage).

![Diagram of the 3081E event unit]

**Figure 14. The 3081E event unit**

7. The MacVEE control, monitoring and development system

A high level of user interaction is required with all the component parts of a multi-processor data acquisition system for the development of its software and the monitoring of its operation. During development, the program code has to compiled and loaded into all the processors, the intermediate results must be displayed, and the system errors reported. During experiment data-taking, powerful but easily-handled interactive tools must be provided to allow any user to master the complexity of the apparatus and monitor its operation.

MacVEE Plus [11] systems, which provide the M68000-based Apple Macintosh Plus computer with direct access to VME and CAMAC, have been adopted for this purpose at the UA1 experiment. The graphics-oriented user interface of the Macintosh, with its pull-down menus, multiple windows and mouse, has proved ideal in this application.
Mac\textsuperscript{VEE} (Microcomputer Applied to the Control of VME Electronic Equipment) allows over 100 Mbytes of external address space in up to 8 VMEbus crates to be mapped directly into the 16 Mbyte address space of the Macintosh. Additionally, Mac-CC dedicated Macintosh CAMAC crate controllers allow the Macintosh direct access to up to 8 CAMAC crates. Each CAMAC crate occupies 64Kbytes of address space when the CNAF (Crate, Station Number, Sub-Address and Function Code) is memory-mapped in accordance with CERN M68K-CAMAC conventions.

In a MacVEE system, the VMEbus master modules and CAMAC Mac-CCs are interconnected by a ribbon cable bus. The bus is driven by an electronics plinth called MacPlinth, which attaches to the Macintosh and becomes an integral part of the computer. MacPlinth maps memory references by the M68000 to internal Macintosh or external VMEbus or CAMAC space in accordance with a memory segment table stored in PROM. In this way the VME or CAMAC access is completely transparent to the user and no special software is necessary to handle the interface.

12 MacVEE systems are currently used in the UA1 experiment for the development of the multiprocessor software and the control and monitoring of the experimental apparatus. The system has proved popular with other researchers, and a total of over 175 systems are now in service.

![Figure 15. The MacVEE system](image)

7.1 The MacUA1 development system

The programming languages used for the UA1 VME data acquisition are Fortran and M68K assembler. Software development, which was originally based on cross-compilers running on the NORD-500 computer, has now been moved to Macintoshes.

The Macintosh application MacSys [12], developed in UA1, allows the editing, the compilation and the execution of Fortran and M68K assembler programs. MacSys is based on a multi-window interactive text editor using the Macintosh filing system and optionally the MacVEE extensions. In addition it can execute position-independent M68K code (that is,
program modules having all their memory references either by register or relative to the program counter) loaded either into Macintosh internal memory or into a VME memory.

The following position-independent programs run under MacSys:

- **M68K Real Time Fortran (RTF) compiler.** The RTF compiler is written in a 'very high-level' language, then translated into a Fortran program itself, and the result of its own compilation is the M68K native compiler that runs on Macintosh. The RTF compiler is compatible with the Fortran 77 standard and includes some extensions oriented to real-time programming and the handling of memory-based devices. For example, the equivalence of a variable to an absolute address is allowed and the common block base address can be dynamically assigned at run time. M68K instructions can be embedded in the Fortran source and the code produced is re-entrant and position-independent. The compiler supports all the M68000/08/10/12/20 family of processors and a variety of floating-point software and hardware options (MC68881 FP coprocessor or software emulation and NS16081 peripheral processor).

![Figure 16. The MacSYS development system block diagram](image_url)

- **M68K assembler.** This was originally an M68K native assembler for a CP/M system, modified to include some of the CERN cross software directives. It produces M68010 position-independent code. No intermediate code is used by the MacSys compilers, because all the library modules produced by RTF contain an entry point description table and the external (subroutine or common block) symbol references in the code itself, allowing linking at run time.

- **Linker.** This is an assembler program used by MacSys to load, link and execute RTF programs. The linker also incorporates a VME memory manager and some tools for the downloading of programs and the control of a VME CPUA1 processor.

- **Libraries.** The libraries includes the RTF run-time main library, the Macintosh toolbox RTF interface and a variety of subroutines for histogramming, graphics and the testing and handling of UA1 VME modules.
- **User programs.** Any user program developed with MacSys can be executed under the control of the linker using the MacSys interfaces to the Macintosh operating system. In addition, MacVEE allows the user and linker code to be moved into any UA1 VME processor and executed locally.

### 7.2 VME RAM disk

The memory of a VME crate can be used by a MacVEE system as a RAM disk to enhance the Macintosh system performance. The UA1 application AMERICA [13] (A MacVEE External Ramdisk Install and Configure Application) includes a MacVEE external disk installer, a VME memory manager and the system drivers to handle the VME RAM disks. Up to twelve VME disks can be mounted in one or more VME crates at the same time.

### 8. The VME data acquisition software

No standard software operating system is used to supervise and coordinate the task execution and the message exchange during data acquisition. Instead, the operating system is embedded in the hardware layout itself. Each processor unit is associated with a preassigned task and the message communication is always implemented by common memory blocks, accessible either through a dual-port memory or a public memory in a VMEbus extended by the crate interconnect (Fig. 17).

![Image](image)

**Figure 17. The shared memory scheme**

Furthermore, the hierarchical structure simplifies the real-time synchronization of all the tasks. The overall system can be regarded as a data flow network in which each element pipes data from an input port to an output one and executes its task whenever the input data are available. A readout unit reads and buffers the data from the detector electronics when an event trigger is pending. An event is built by the event manager when the event data are ready in the parallel readout buffers, and an event unit processes an event when its buffer is full.
The VME data acquisition software is written in M68K assembler and Real Time Fortran (RTF). All the M68K processors run a resident monitor (CPUA1MON) that provides the basic functions to handle the module input/output peripherals, program downloading and debugging, and the basic interrupts.

All the software modules are position-independent (only program counter relative references are used) so that the code can be moved in memory without re-linking. This convention simplifies downloading to remote processors and software sharing.

The data acquisition software consists of the event manager, the readout supervisor and the programs in the event units.

8.1 Event manager

The event manager is an assembler-written program whose main functions are:

- **Hardware recognition.** Identify and initialize all the event data buffer dual-port memories. Build the hardware modules list and check the list against the minimum required set specified in a configuration table.

- **Initialization.** Initialize the readout supervisor event directory and the hardware readout configuration table. Start all the active event units.

- **Main loop.** Wait for a data acquisition command coming either from the terminal or from the NORD data communication mailbox.

- **During the data readout.** Accept any event unit request and create an event request list. When an event becomes available, read the position and size of all the detector data blocks from the event directory. Via multiple DMAs, broadcast a full event to the event unit buffers mapped at a common VME address on the event data bus. When an event is processed by the primary event unit (the mass storage one), communicate to the readout supervisor to free the parallel readout buffer of that event.

8.2 Readout supervisor

The readout supervisor is an assembler-written program, which communicates with the event manager through a common memory. Its main functions are:

- **Recognize** all the parallel readout units. Initialize the event directory and select the units to read in accordance with the table loaded by the event manager.

- **Wait for a trigger,** then start all parallel readout units. Wait for the completion of data input, pulling the status of each readout unit. Monitor readout errors such as timeout, no data DMA error, data overflow, bad word count and other checks according to the type of input unit.

- **Enter** the event parameters into the event directory and wait for the next event.

- **Wait** for event manager commands (stop, reset, free buffer).

8.3 Event unit

Each event unit runs a Fortran program processing the full event data. Event manager synchronization is maintained by a few assembler-written subroutines for functions such as: set the buffer parameters, get one event, release an event buffer. The main event units are:
- **168E-NORD tape storage.** By means of a VME/VMX to CAMAC data link the read events are sent to the 168E emulator farm before being transferred to the NORD data acquisition computer.

- **3081E filter.** The program handles the status of six 3081E emulators interfaced to VME, switching the event manager data path into the first free emulator memory, monitoring the processor execution and sending the data output to the mass storage device (video disk or NORD link).

- **ZPHYS.** This event unit is used to monitor the detector data by accumulating histograms and statistics. The data display is performed by another processor accessing the histogram memory via VME.

- **Event display.** The event unit processes the information from the track detectors and produces event displays on a colour monitor driven by a VME graphics module.

- **Calorimeter monitor.** This event unit processes and displays the calorimeter data as for the main event display.

9. **Conclusion**

The UA1 data acquisition system illustrates a number of evolutionary trends which are changing significantly the ways in which physics data are handled in an on-line experiment.

The vastly increased number of channels in large complex detectors imposes a configuration in which signals are digitized and multiplexed at their origins. To extract the maximum amount of information from the detector signals, they are digitized with fair resolution so that physical parameters can be determined by numeric operations.

As a result, a very large data volume is generated per event, and it becomes necessary to implement a high degree of intelligent parallelism in the data readout system and to create a sophisticated multi-level trigger to achieve an acceptable event rate. Such a parallel processing system can be implemented in a flexible and cost-effective manner using the industry-standard VME/VMXbus framework.

The programming, control and monitoring of such a complex distributed processing system presents new challenges which can be met by the application of the latest generation of low-cost personal computers, having a powerful graphics-based user interface and direct access to the VMEbus and CAMAC systems.

The use of the industry-standard VMEbus allows performance improvements to be incorporated with a minimum of special development as microprocessor technology evolves. The transparency of the VME communication protocol simplifies the configuration of the complex data acquisition structures required in a large modern experiment. Finally the modularity of the VMEbus architecture allows the required data flow structure to be elegantly synthesized by the physical configuration of the hardware itself, and to be readily upgraded as requirements change in the future.

**Acknowledgements**

The development of the UA1 VME data acquisition system was supported by C. Rubbia.

The software for the data acquisition system was written by members of the UA1 collaboration, while the hardware was developed by members of the CERN EP-ELD and DD-ED groups and the institutes collaborating in the experiment.
The bulk of the system software was written by S. Cittolin, M. Demoulin, W.I. Haynes, W. Jank, P. Petta, J.-P. Porte, D. Samyn and H. Von der Schmitt.

The MacVEE system was developed by B.G. Taylor.

The VME REMUS branch driver and VME-VMX DMA module were developed by C. Engster and L.G. van Koningsveld (EP-ELD).

The 168E and 3081E VME interfaces were developed by A.Fucci, P. Gallno and B. Martin (DD-ED).

The VME crate interconnect was developed by E. Pietarinen (Helsinki University).

The VME parallel input/output module was developed by B. Boget and J. Dufournaud (LAPP).

The Streamer tube readout (STAR) was developed by S. Centro, D. Pascoli and P. Rossi (Padova INFN).

The VME interrupt vector generator was developed by A. Arigon and P. Farthuat (Saclay).

References


- E. Pietarinen, The VME Crate Interconnect Module, Helsinki University, Finland.
- The VME/VMX bus CPU module, DSSECPUA1 PHM 1.0 June 1984, Data Sud Systems S.A.
- The VME/VMX bus Dual Port Memory, DSSECDSRX June 1984, Data Sud Systems S.A.
- VMECPU 020, Robcon OY.


- M. Demoulin, CPUA1 Monitor, UA1 technical note 85-36.

MODULAR ARCHITECTURES FOR ON-LINE SYSTEMS

A. Vascotto
CERN, Geneva, Switzerland

1. Introduction

In this lecture I will describe some development work being done by the On-line Computing Group at CERN, in the field of data acquisition systems. Originally I had been requested to present a case study of a LEP data acquisition system; unfortunately there is not such a case yet to discuss, because the systems are being designed now and are still at an early stage of development. My subject though is very near to the one proposed, and should be understood as a case study of a development work which is still evolving.

I shall describe the model on which is based the architecture for future on-line systems and give the motivations for the choices that have been made. I shall also present a more detailed description of some of the packages that are being currently prepared; it will be shown how some of the architectural aspects originated from the aim of making the packages modular and adaptable.

2. Data acquisition systems

In order to give a clear boundary to the subject of this lecture, it is worth specifying that I shall only discuss the software for the on-line computer or computers. Normally the term "data acquisition system" also includes the front-end electronics where the data collection takes place, and the processors that may select or modify the events before they are read into the on-line computer. In the context of this lecture though, the meaning of "data acquisition system" (and the one of "on-line system" used sometimes alternatively) concerns exclusively operations performed by the computers charged with the running of the experiment.

The tasks that are traditionally accomplished by a data acquisition system are obviously related to the event processing: data read-out and event formatting, data recording, monitoring the quality of the data by analysing a sample of events.

Other tasks are also very important in the management of the experiment: the organizing of the activities concerning a coherent set of events (run control), the re-playing of recorded data to pass the events through a more complete analysis chain or to try out new selection algorithms on recorded events.

There is also a whole range of tasks devoted to the control of the apparatus, such as setting-up procedures, test procedures, and calibrations.

Of course the on-line machines are expected to continue providing the general services of a computer system: program preparation, communication services, and so forth.

People have been even installing on the on-line computers facilities that were, in the past, only reserved to main-frames, such as data-base management systems.

Data acquisition systems with these features exist and are successfully used. I shall explain in the next section why it was felt that, for the LEP experiments a new design was necessary, based on a new architectural model.
3. The reasons for a new design

There is a continuous evolution in the high energy physics experiments. As the knowledge progresses, new fields of research open up, that require always larger detectors, more electronics, more powerful computers and more people to run the experiments. This is a general trend that has always been seen; it certainly holds now, in the case of the LEP experiments.

As a consequence, new trends have shown up concerning the on-line systems. The following sections will discuss some of these new trends.

3.1 Computer technology evolution

I shall not amaze anybody if I say that computers are getting cheaper and more powerful at a rather constant pace. The consequence is that experimental groups can now afford to put several computers in the experimental hut.

The availability of local area networks at reasonable prices makes it possible to form with these computers a rather homogeneous complex, even though not all the problems are solved yet, as we shall see.

3.2 Shift of functionalities

The on-line computer used to be the only processor in the read-out chain; therefore it had to organize the event building by driving the detector electronics.

Now the front-end electronics contains powerful microprocessors that take all the burden of reading the detectors and formatting the events by collecting the bits and pieces. This implies also the existence of a lot of electronic buffers to temporarily store several events; therefore the rate at which the events are extracted is more uniform and a short response time to triggers is no longer necessary.

On the other hand, the management of all the processors in the front-end electronics becomes the crucial point: initial loading, diagnostics and communications are now key issues.

3.3 Experiments' growth

The collaborations that will run the LEP experiments are much larger than used to be those of the previous generation. Their requirements are accordingly more complex and more demanding. Many people take part to the decision-making process, and eventually the most advanced technologies are chosen in every field, including on-line programming.

A product that does not fit exactly the requirements established by the system designers has no chance of being adopted. No compromise is possible.

On the other hand, the collaborations do have manpower available for the development of the on-line system. This is a big change from the (not too far) past, when experimental groups were eager to obtain a turn-key data acquisition system for their on-line computers.

4. A model for a different approach

The new approach for the design of an on-line system targetted to the LEP experiments is the outcome of the analysis, previously described, of the trends, and of the consideration of the criticism formulated on existing data acquisition systems.

The approach is based on a hierarchical model.
In this model, the bottom layer is constituted by the infrastructure: services normally provided by operating systems or otherwise commercially available products, such as scheduling primitives, input-output services, communications. In the infrastructure one can include also services that are not always found and have thus to be provided: FASTBUS or VME access, window management systems, and so on.

Above the infrastructure there is the layer of the on-line services. These are the building blocks for a data acquisition system and provide the basic functionalities that are common to any such system.

The on-line services are provided by agents that may simply be subroutine packages or more complex suites of programs; in any case all their features must be accessible via procedure-call interfaces.

The real challenge has been to determine whether there was such a common ground on which the LEP experiments could agree, and decide what the services had to be. Example of services are the buffer manager, the message system, the information manager, the process control and so on.

At the top of the hierarchy there are programs that cover the traditional high-level functions such as organising the data recording, monitoring the apparatus, controlling the experiment and so on. They use the on-line services as a set of tools to perform actions on their behalf. These programs are the visible part of the on-line system and must be suited to the needs of the physicist on shift. Each experiment will have to provide these programs, tailored to the peculiarities of the procedures they have established.

From what precedes, it is clear that the On-line Computing Group decided to concentrate its effort on the intermediate level of services. Two major goals preside this development effort: modularity and openness.

- Modularity is intrinsically given by the approach of on-line services. Services can be independently chosen and integrated to obtain the desired style of customization. But we also want modularity to extend deeper in the system: each service will have to be designed in such a way that, whenever a particular choice in a module would constrain a given style for a function visible to the end-user, then the module must be made replaceable.
- Openness is nothing else than another facet of modularity: the interfaces between modules must be documented and accessible, as well as the data structures used by the modules. This guarantees the possibility for the system designers to adapt the behaviour of a service to the requirements by replacing modules.

Some of the services that are currently being designed will be described in the next sections.

5. The buffer manager

One of the main tasks of the on-line system is to monitor the quality of the data collected by analysing a sample, as large as possible, of the events being recorded. The function of the buffer manager is to allocate memory space for the events being read, to keep track of their locations and to grant access to these events to the monitoring processes.

The fact that events are buffered has also the effect of smoothing the differences between the input rate and the output rate. We have already said that for the LEP experiments this is much less important, since the buffering occurs in the front-end electronics and the rate seen by the computer is low and uniform.

In the architecture of the buffer manager we have introduced a number of new concepts. Events are read-out by one or many processes, called event producers. Producers belong to the top layer in our architectural model, as monitoring processes (consumers) do. The producers declare the events to the buffer manager, where they are organized into streams.
A stream does not imply a movement of the events in physical memory: actually events are never copied during their life time, except when strictly necessary or specifically requested. The concept of stream concerns the modifications that an event may undergo while in the system. A stream is composed of stages; an event is injected into a certain stage and will have to perform a number of mandatory stage transitions, every time it is modified by specific processes. Examples of modifications to be applied to events are: compacting the data, adding pieces of information coming from other sources, and so on. The modifications are operated by programs that access events in the same way as monitoring programs do, but have special privileges.

Of course a stream, in the simplest case, may be constituted by one single stage.

Another interesting concept is that of a port: a monitoring process obtains events from a stream by attaching to a port, which is essentially an output stream of the buffer manager. A port is attached to a given stage on a given stream; some selection criteria may be applied to the events dispatched to the port, therefore the port stream need not be identical to the main stream to which it is attached.

The setting up of the ports and their profile is done by programs via subroutine calls.

With the concept of ports we achieve a major goal: we decouple the selection of events from the programs that are monitoring them. The selection is defined independently beforehand; the only thing that the monitoring program sees is a stream of sequential data, as if it was reading a magnetic tape. This is essential to allow large analysis programs, designed to be used off-line, to be included in the on-line environment.

Several processes can attach to a port, provided the selection criteria and the monitoring mode are the same for all of them.

The port profile includes also such characteristics as whether all the events have to be dispatched, or a fixed percentage, or as many as possible.

Events are kept in the system as long as there is enough space for incoming events; when no more space is available, events that are in their final stage are discarded, provided they are not currently booked or in use. In this way the monitoring processes have the largest probability to find something useful in the buffer, but event producers are less favourably treated, because of the delay they may incur when a clean-up is necessary.

The whole buffer manager is organized as a set of shareable routines; there is no supervisor process. The selection criteria to be applied on the ports can either be chosen among a number of off-the-shelf recipes, or provided by the user via shareable routines.

6. The human interface

Under this heading we collect those services that allow the users to interact in a homogeneous way with programs throughout the system. It is worth stressing here that by “system” we mean a complex configuration of computers, with some sort of network interconnection. Today’s operating systems provide you with facilities that help dealing with such configuration: virtual terminal for remote sessions, remote job submission and load balancing, file sharing. Still the system is not seen by the user as one single black box: a certain knowledge of the configuration is necessary. Our goal is to hide the configuration details into an interface that on one side is capable to deal with a loosely coupled system, on the other presents to the user a coherent and homogeneous front.

Another major concern is the dichotomy between the programmer writing his program using a development system, possibly back home, and an operator dealing with the program once it has become part of a larger complex. A standardization is needed on both sides. On the programmer side, rules should be obeyed in order to allow the future integration of the program into the system in a cooperative fashion. On the other side, the user would like to deal with any program in the same, easy, homogeneous way.

150
Our aim is to give the framework in which these problems can be solved, without imposing rules or styles that necessarily only a minority of users would have appreciated. This goal has been achieved by designing an architecture that is open enough to allow the customization of the style in the human interaction, by plugging different modules into a system that takes care of all the underlying technicalities.

The problem addressed is the one of the control, from a terminal, of a program to be run or running somewhere in a cluster of machines. Incidentally, the terminal cannot be dedicated to this single activity: we want to be able to deal with several programs at a time from the same terminal.

The heart of the architecture we have designed is the Terminal Control System (TCS). The TCS is a process that takes care of the activity going on at a terminal. For each terminal there is one copy of the TCS on control.

The TCS is a multithreaded program, stimuli driven, with a small kernel called event manager (in this context, events are the stimuli), which controls the execution of a server routine for each event that may occur. The stimuli or events are given either by the activity of the terminal or by messages on the network.

To describe how the TCS works, I will start with an example, namely the normal terminal input—output performed by a program. Within the TCS there is a server routine in charge of routing the terminal traffic of all the programs connected to that terminal. The characters typed at the keyboard trigger the TCS that jumps into the server routine, which in turn will send the string formatted into a message on a suitable communication channel.

The user's program is linked to a shareable library containing the partner of the TCS server; this receives the message and dispatches the string on the program's I/O channel.

This example is sufficient to show some of the features of such an architecture.

First of all, the user's program need not issue special calls in order to perform its I/O, therefore the program can run in stand-alone mode, when not yet inserted in the system. The same program, when it joins the party, will be able to utilize the TCS path with no modifications.

Secondly, the user's program is totally decoupled from the hardware it is actually using (the terminal, mouse, tablet, etc.). This offers the possibility of using one terminal to communicate with several programs at the same time. We also have the freedom to organize the screen as we like, with the usage of a window management facility either native in the terminal hardware or simulated within the TCS.

Furthermore, the idea of pairs of servers (one within the TCS and one linked to the user's program) can be used for other purposes than the terminal I/O. Take the example of a menu package. It can be run in the user's program and just issue strings of characters to output its layouts. Alternatively, owing to the TCS architecture, it can be split in two parts. One is attached to the user's program and consists of the functional section, where actions are taken according to the operator choices. At the other end, in the TCS, there is the part that creates the menu displays on the basis of requests coming from the partner menu server (in the user's process). The TCS part may depend on the terminal type, in order to use more effectively the hardware available. Moreover, the machine where the user's program is running (and doing "real work") is offloaded.

Another important aspect of the human interface is to provide a uniform way of controlling the execution of a process. A server in the TCS provides facilities to the interactive user to start a process, stop it, or re-enter the communication phase. The server has access to shareable tables containing the location of each process known to the system. The partner of the TCS server is in this case another process which is entitled to start and stop processes in the machine where it is running. Of course, if the execution of the user's process is to be altered, special routines must be called at given checkpoints, to allow for branching out from the main flow of the program.
7. The error and message utility

A modern operating system provides you with facilities to format, output and log error messages. Still the needs of an on-line system are quite peculiar and the facilities provided by the operating system are not sufficient.

The problem is similar to that previously described in the section on the human interface. Programs are often used to make checks, calibrations or to verify running conditions. When they discover something abnormal, they signal it to the operator on shift. Sometimes a message is issued to show that everything is running smoothly. When the program is tested by its author, the messages are issued to an interactive terminal. It is clearly not tolerable that each program continues doing so when it participates with possibly hundreds of other programs to the running of the experiment; on the other hand the programmer does not know beforehand where the messages will have to go.

Therefore, it is not unusual to have a central message server (and existing on-line systems do have it) which receives all the messages, formats them according to the information contained in a disk file (not in the issuing program) and dispatch them to the suitable device or devices.

The advantages of such a system are multiple.

- The program provides only the variable part of the message, whilst the text and formatting information are kept elsewhere; it is therefore easy to modify these without changing the program.
- The centralized control of the message flow allows for the collection of message statistics and for the suppression of too frequent messages.
- The messages can be dispatched to more than one destination, such as a status display, a printing terminal, an error log file, etc. The decision on the destination of messages can be taken when the program is integrated into the system.

In the new message facility (called Error and Message Utility: EMU) we have introduced new ideas to enhance the modularity and the flexibility of the package. By analysing the functionalities required from a message system, we have reached the conclusion that there exists a sequence of independent actions to be performed on every message.

- The first is to associate a message, received from the issuing program, with the formatting information, therefore creating a text string, meaningful to the human operator.
- The second is to dispatch the message to its destinations, according to given rules as we will describe below. It is worth stating here that the message has no association whatever with its own destination, because the latter may change according to the running conditions. For instance, you may want to get messages coming from a given program on your interactive terminal, while you are checking a particular detector.
- The third action consists of outputting the message to its destination, which may be a log file, a window on a screen, a statistical record and so forth.

These three actions can remain distinct and are thus performed by different modules of the EMU, called the decoder, the router, and the presenter.

One decoder must be installed in each machine; one single router will serve the whole system; one presenter is needed for each destination device.

We have already said that the routing information is not present in the message at the decoder level. However the message is assigned there some "properties". A property of a message is a string associated permanently with it, which is meaningful exclusively in the context of a given experiment. The system designers are supposed to define such properties in a way that can be used for selecting useful streams of messages. For instance, message properties may be "central detector" or "severe" or "red". The router will dispatch the messages according to a table that maps properties onto destinations, such a table being set up either interactively or by programs via routine calls. The dispatched messages form streams towards the presentation modules.
Various means of communication will be used in this system. The user's programs talk to the decoders via mailboxes; the decoders send their streams to the router via a network channel, the two partners not necessarily being on the same machine. The presenters may just be routines linked with the router, provided all the devices used are connected to the computer where the router is running.

The presentation modules deal with the final destination of a message stream: it can be a log file, or a display device, or a statistics-gathering program; the users can attach their own modules, and decide the way the messages will be shown.

8. Other services

The services described so far have been implemented first, because they constitute the fundamental parts of a data acquisition system. Other services will have to be provided as well, such as:

- Run Control, whose purpose is to coordinate the activities during the data acquisition periods (usually called runs).
- Interactive Task Synchronization: a framework within which different tasks can cooperate, executing calibration procedures according to pre-established sequences.
- Occurrence Signalling Package: processes may declare to be interested in given events and wait for them to occur; they continue processing when the occurrence is signalled (by another process).
- Information Management. The purpose of such service is to organize and facilitate the access to the information produced by all the programs in the system.
- Resource Management: the allocation mechanism to grant an orderly access to non-shareable resources.

Some of these items are currently being designed.

9. Conclusion

We have shown that several of the services described here have been introduced in order to provide a framework in which the activity of each single program can be supervised: the goal is to allow for an easy and coherent integration of a given functionality with the other activities necessary to run an experiment.

Another problem being addressed in some cases is the complexity of the multi-computer configurations. The aim here is to present the system to the end-user as a whole processing entity, the details of the CPU's configuration and interconnection being hidden by a layer of software that takes care of accessing the resources wherever they are.

These aspects of the development we are engaged in are related to the fact that most of the operating systems, being originally designed to run on a single CPU, do not provide facilities to cope with this kind of problems.

However, if multi-processor systems will continue to be successful as they are beginning to be, the situation is likely to change: these problems will become more visible, new operating systems, targeted to distributed computing, will include functionalities similar to those our research work is trying to implement now.
SOFTWARE FOR DISTRIBUTED SYSTEMS

J.C. Vermeulen

NIKHEF-H, Amsterdam, The Netherlands

Abstract:

Concepts needed when programming multiprocessor systems, with emphasis on systems where each of the processors executes its own instruction stream, are discussed.

1. Introduction

Using Flynn's taxonomy (1), two main types of distributed systems can be distinguished:

- SIMD: Single Instruction Multiple Data stream: a number of processors, that execute synchronously the same instruction stream, e.g. the ICL DAP (2) or the NCR45CG72 Geometric Arithmetic Parallel Processor (3). The latter consists of a 84-pins integrated circuit, containing 72 1-bit processors in a mesh-connected 6 x 12 arrangement:

```
  1  2  3  4  5  6  7  8  9  10 11 12
  13 14 15 16 17 18 19 20 21 22 23 24
  25 26 27 28 29 30 31 32 33 34 35 36
  37 38 39 40 41 42 43 44 45 46 47 48
  49 50 51 52 53 54 55 56 57 58 59 60
  61 62 63 64 65 66 67 68 69 70 71 72
```

Each processor contains a 128-bit random access memory, 4 1-bit registers, a full adder / subtracter and is connected with its 4 nearest neighbors (via registers). There are 20 control lines for the integrated circuit: 7 address lines and 13 lines carrying the instruction to be executed. 5 simultaneous operations are coded with these lines (e.g. move from "east" in the EW register, move from "north" in the NW register, etc ...). The NCR-chip is described by NCR as a "systolic array", but is in fact an implementation of Unger's Spatial Computer (4). The power of this architecture stems from the fact that many processors can be used, while communication between processors is fast (NCR-chip: 1 bit per machine cycle = 100 ns). The fact that all processors are executing the same instructions has of course consequences for the algorithms to be used. However, the software will basically consist of one sequential program, for which the
data should be structured in an appropriate way. Example applications of SIMD processor arrays are: digital image processing (5) and lattice gauge calculations (6). Application of SIMD processor arrays in trigger processors for high-energy physics experiments are certainly worth consideration.

- **MIMD**: Multiple Instruction Multiple Data stream: a number of processors, that execute asynchronously their own instruction streams, e.g.:
  1. a number of processors (CPU + memory) connected with each other via a local area network (a *loosely coupled system*),
  2. a number of CPU's, having access to memory modules via a switch system (a *tightly coupled system*),
  3. a number of "transputers" (7) connected to each other via special (fast uni-directional serial) links (a *moderately coupled system*), e.g. in the following configuration:

![Diagram of a MIMD system](image)

Note the similarity of this arrangement with the arrangement of processors in the NCR SIMD array. However, here we have 16- or 32-bit processors, that each execute their own instruction stream and that communicate with other processors in the array when this instruction stream dictates them to do so. Furthermore other arrangements are also possible. The transputer is manufactured by INMOS in a 16-bit version (type T212) and a 32-bit version (type T414). One integrated circuit contains the CPU itself, 2 kB of fast RAM, and 4 serial link interfaces, where each interface services an outgoing and an incoming fast (20 Mb/s) asynchronous serial link.

The software for an MIMD system will consist of a *number of processes*, at least one per processing element (PE) available. In many cases these processes have to *communicate* with each other and their actions should be *synchronized*. As can be inferred from the examples, communication between two PE's can proceed via shared memory or via connections like dedicated serial links, networks or dedicated parallel links. When no shared memory is present, only the exchange of messages is possible. This is referred to as *message passing*. Using message passing shared variables can be simulated, if desired.
In this contribution the emphasis is on the support needed when programming MIMD systems, consisting of conventional processors. The support facilities can be available to the programmer:

1. via system calls, issued from processes written in a conventional language (C, Pascal, FORTRAN, etc.). To make this possible a distributed operating system is needed,
2. via facilities in the programming language used, as provided e.g. by Ada and occam. Also here a distributed operating system is needed, but this is accessed via the language and not directly via system calls.

Of course, when no support is present, or the support available does not provide adequate service or speed, the facilities required have to be implemented by the programmer himself.

For the development of software often the host-target approach is used, which means that program development is done on a host computer (often a general-purpose personal computer, minicomputer or mainframe). The executable code is "down-line loaded" to the target processors that execute the code. The file-system of the host is often accessible by processes running in the target system, while also terminal I/O can proceed via the host. Of course the host should provide the programmer with a decent programming environment.

2. Fundamental concepts needed when programming MIMD systems

2.1. States of processes

We will allow an arbitrary number of processes per PE. These processes can each be in one of four states: (i) executing (only one process per PE), (ii) ready for execution (each PE maintains a ready queue; each process ready for execution on that PE has its own entry in the queue), (iii) blocked, (iv) inactive (not yet loaded) / terminated. Two basic strategies for scheduling the execution of processes on a given PE exist: (i) preemptive scheduling: the execution of a process can be temporarily stopped by a scheduler, in order to give other processes a chance to to execute, and (ii) nonpreemptive scheduling: a process that executes, only stops with execution by actions taken by the process itself, which result either in blocking of the process or in termination of the process.

The use of preemptive scheduling gives rise to quasi-parallel execution of multiple processes on one PE: for the user of the system the processes run seemingly simultaneously, but the CPU can execute the instructions only one after the other. A rescheduling operation can in principle occur at any arbitrary moment, so the software should be constructed such that this can be tolerated.

2.2. Organization of inter-process communication

There are two possibilities (8): (i) use of shared variables, and (ii) message passing.
2.2.a. Shared variables

Often it is necessary to prevent that two processes are using the shared variables at times that this is not allowed. An example of this is the producer - consumer problem: PE A produces data that should be consumed by PE B. PE A fills a buffer, that is emptied by PE B. PE A can only produce data when there is still room in the buffer, PE B can only consume data when there is data in the buffer. Both PE's should have a consistent view of what is produced and consumed in order to have this working properly. This can be organized such that the processes will need to have the possibility to be in a state, a so-called critical section, in which they have exclusive access to parts of the shared memory (during this state data will be either produced or consumed in this example). The state can be signalled with a shared variable, for which it should be possible to read and to modify the contents in an uninterruptable way. For this purpose often an undivisible read-write cycle is provided.

Examples of undivisible read-write cycles ( { ..... } signifies an undivisible read-write cycle):

- Test-and-Set (TAS):
  \[
  \text{TAS (var) = \{} \text{ test var ; set status bits in condition register of the processor ; var := 1 } \\}
  \]

- Compare-and-Swap (CAS) (var is a shared variable ; var1 and var2 are local variables):
  \[
  \text{CAS (var1, var2, var) = \{} \text{ if var = var1 then} \\}
  \text{ set status bits in condition register;}
  \text{ var := var2}
  \text{ else}
  \text{ set status bits in condition register } \}
  \]

- Fetch-and-Add (9):
  \[
  \text{F & A (var, e) = \{} \text{ temp := var ; var := var + e ; return temp } \\}
  \]
  (var is a shared variable, temp and e are local variables)

With the TAS instruction semaphores can be implemented. A semaphore is a special type of variable, for which only the following two operations can be executed:

\[
\begin{align*}
\text{P(s) => \{} & \text{ if } s > 0 \text{ then } \\
& s := s - 1 \\
& \text{ else } \\
& \text{ block execution } \}
\end{align*}
\]

\[
\begin{align*}
\text{V(s) => if } & \text{ a process is blocked after a call to P(s) } \\
& \text{ then deblock this process } \\
& \text{ else } s := s + 1
\end{align*}
\]

s can only have a value of 0 or 1 in the case of a binary semaphore. In the case of a general semaphore or counting semaphore s can be a positive number. In stead of blocking of the execution in the case of P(s), s can also be regularly tested until s > 0.
Use of the TAS instruction for the implementation of a binary semaphore:

\[
P(s) \Rightarrow \text{TAS}(s); \quad \text{if status bits indicate that } s = 0 \\
\text{then} \\
\quad \text{execution of critical section can start} \\
\text{else} \\
\quad \text{execution of critical section not possible} \\
\]

\[
V(s) \Rightarrow s := 0 
\]

Remark that \( s = 0 \) signals the possibility of execution of the critical section, that blocking of the execution of a process is often done by the operating system, and that only for \( P(s) \) an undivisible read-write cycle is needed.

The instruction set of the 68000 processor family contains a TAS instruction (10). The processor indicates with a low "address strobe" signal during two consecutive bus cycles, that a read-modify-write cycle, which is an undivisible read-write cycle, is executed. The instruction tests the contents of a byte and always sets the most significant bit of the byte to 1. With this instruction only binary semaphores can be implemented.

The CAS instruction can be used for changing "safely" e.g. shared pointers or shared counters, as demonstrated in the next example, where CNTR is a shared counter that indicates how many times a certain operation by all PE's in a system has been performed:

\[
\begin{align*}
\text{temp1} & := \text{CNTR}; \\
\text{repeat} \\
\text{temp2} & := \text{temp1} + 1; \\
\text{until} \\
\text{CAS (temp1,temp2,CNTR) indicates that} \\
\text{temp1} & = \text{CNTR}, \text{which means that CNTR has} \\
\text{been set equal to temp2} \\
\end{align*}
\]

The CAS instruction increments CNTR as soon as possible. Remark that straightforward adding of 1 to CNTR is not possible, because an other PE can modify the contents of CNTR after fetching the contents of CNTR, but before writing the result back. The Fetch-and-Add instruction could also have been used in this example. An other possible application of the CAS instruction is updating of pointers in a shared linked list for those cases for which only one pointer has to be modified. However, often two pointers have to be modified simultaneously, e.g. when a record has to be added to the list.

The instruction set of the MC68020 microprocessor (10) contains the TAS instruction as well as two types of CAS instruction: the CAS instruction as discussed and the CAS2 instruction. With the CAS2 instruction e.g. two pointers in a linked list can be modified in one undivisible read-write-read-write cycle:
CAS2(testvar1, testvar2, update1, update2, var1, var2) =
   { if testvar1 = var1 \textbf{and} testvar2 = var2
    then var1 := update1 ; var2 := update2 ;
    set status bits in condition register ;
   else
    set status bits in condition register }

Processes running in a single PE might not have access to the whole memory accessible by the PE. In that case a so-called \textit{Memory Management Unit (MMU)} is used, that translates each address output by the CPU, the so-called \textit{virtual addresses}, into \textit{physical addresses}. The MMU checks that the addresses are valid for a given process, and when this is not the case, generates an interrupt request for the CPU, that should be handled by the operating system. A memory management unit makes it impossible to write in arbitrary memory locations, and thus can prevent that the code of a process itself and/or code and/or data of other processes is overwritten. However, it can also make communication via shared variables between processes executing on the same CPU impossible, if the operating system does not adjust the MMU such that processes have access to the same memory partition.

\subsection*{2.2.b. Message passing}

The basic principle of message passing is that a given process A sends data to an other process B. This can be done by letting A directly write to an interface or shared memory that is used to provide B with data sent by A. This method is always applied at the lowest level of the system software; however, it is desirable to have the possibility for the user to send the message to e.g. a \textit{port} or a \textit{mailbox} with a certain symbolic name. The names of e.g. mailboxes should be associated with the interfaces/memory locations to be used for inter-process communication. There are two possibilities: (i) all possible ”software communication channels” are specified \textbf{before} execution \textit{(bind-at-compile-time)}, or (ii) when starting a process the correspondence between interfaces/memory locations and names of e.g. mailboxes is unknown. The operating system must then be able to determine the correspondence \textit{(bind-at-execution-time)}. A \textit{name server} can deliver this information. Advantage of this method is that the locations where processes are loaded can change during execution, i.e. processes can be moved around the system.

Depending on the operating system or language used, after sending the message the sending process can be blocked until a reply has been received or the sending process can continue execution. Although until now it has been assumed that only two processes are communicating at a time, it is in principle possible that a message is sent to several other processes simultaneously; this is called a \textit{broadcast}.

When receiving a message the following possibilities exist: (i) a process is blocked when trying to "receive" a message until one has arrived, (ii) a process can inspect whether a message has arrived, but is not blocked when no message is available, (iii) arrival of a message leads to interruption of the
execution of the executing process and causes the start of a special routine. This is comparable with the effect of an interrupt request.

Messages can be stored temporarily by the operating system or by the user until the message can be processed. In that case the buffer used must be large enough, or an error message has to be generated when the buffer space is too small. Time-outs can, in the case of blocking communication, restrict the length of the interval during which a process is blocked and cause e.g. an error message if no message or reply has been received during a certain interval.

Deadlock can occur if one or several processes wait for actions of other processes, while one or several of the latter processes wait for actions of the first group of processes ("deadly embracement"). With deadlock that situation is meant, in which a complete computer system performs no useful work at all; lockout occurs if one or several processes are not able to "wake up", while other processes are still executing. It should not be possible that deadlock / lockout occurs : the "liveness" of a process should be guaranteed. This means also that the communication between processes should be inherently "safe", such that in no instance a deadlock / lockout condition can arise.

The method used for inter-process communication plays an important role with respect to the probability that deadlock / lockout can occur, which is in general smallest when using message passing under the following conditions : (i) the sender is blocked until a reply is received, (ii) the receiver is blocked when it expects a message and when no message has yet arrived, (iii) use is made of time-outs.

Communication using blocking message passing is for the user similar to using procedure calls. When inter-processor communication has the appearance of real procedure calls this is referred to as a remote procedure call implementation (11). The term client - server model is often used in connection with blocking message passing : a client process issues a request for a service to a server process and waits until the server process has completed its task. Protection of inter-process communication may be important, e.g. in a multi-user environment. For message passing a possible technique is to use so-called capabilities as part of the messages (12).

2.3. Allocation of processes to processors

Two situations can be distinguished : (i) static: all processes are loaded during a loading phase, after the user or support software has decided where the processes should go, (ii) dynamic: processes are loaded when needed; the operating system or user processes indicate where the processes should be loaded. It is sometimes possible to let the operating system move processes around in the system, when one or several PE's become overloaded (load balancing ). For real-time systems in most cases it is unavoidable that the user has to decide where the processes should be loaded, because each PE will be connected to specific interfaces to the external world. In general it can be stated that when a
large problem has to be tackled with a multi-processor machine, the algorithm to be used should be adapted to the architecture of the machine. This means that the user and not the operating system or the support software possesses the knowledge of where the processes should go, so in many cases the user has to specify the distribution of the processes.

2.4. Real-time aspects

Real-time software must be able to react on external and internal events. These events are detected by the occurrence of hardware or software interrupts (exceptions) or by inspecting regularly certain status bits (polling). Polling is easy to implement for a user, when the relevant status bits can be directly accessed. An exception can be seen as a message for the process that should handle the exception. The three different possibilities when receiving messages are applicable (i.e. blocking, non-blocking inspection, interrupt-like). A complicating factor is that external interrupt requests often have to be serviced within a limited amount of time. A process, servicing an interrupt request, should then be scheduled with a high priority. There are now two possibilities: (i) the operating system takes care of the scheduling operation after a jump to the operating system due to the occurrence of the interrupt request, or (ii) the scheduling operation takes place by a direct jump to the service routine. A disadvantage of the latter possibility is that the operating system is not able to keep track of what is going on in the PE. The speed of interrupt servicing, however, is maximal. Note that problems can occur when preemptive scheduling is used, that making use of operating system services is not allowed during interrupt servicing and that this method cannot be applied when using memory management hardware, because this hardware has to be adjusted correctly by the operating system to make execution of the interrupt service process possible.

Factors influencing the response time of a MIMD real-time system are: (i) activities of the operating system (preemptive scheduling, updating of timers, activities needed for ongoing inter-process communication), (ii) other events possibly occurring during the servicing of an event, (iii) the influence of eventual use of inter-process communication during the servicing of an event, and (iv) the number and the nature of the instructions to be executed to generate the response, assuming that the first three factors can be neglected.

2.5. Debugging

Debugging in an MIMD environment is in principle a difficult task due to the fact that processes can influence each other via the communication mechanisms available. The timing of inter-process communication can be critical, so that the mere introduction of debugging software can "remove" the bug looked for. In any case, in addition to "conventional" interactive symbolic debugging, it will be desirable to have the possibility to inspect and to influence the states of processes loaded in the system. In general it can be stated that debugging in such an environment has until now received only limited attention.
3. Examples

3.1. Ada

The Ada programming language (13) makes it possible to use multiple processes (Ada terminology: *tasks*), which can communicate with each other via shared variables or via the "rendez-vous" mechanism.

A task should be declared as such by its *parent*:

```ada
procedure PROC1 is
    task AAA;
    task BBB;
    task CCC;
    task body AAA is
        begin
            ...........
        end AAA;
    task body BBB is
        begin
            ...........
        end BBB;
    task body CCC is
        begin
            ...........
        end CCC;
    begin
        -- this part of the procedure must be present;
        -- the three tasks are active after execution of the begin statement
    end PROC1;
```

How the tasks are scheduled is not specified; when using shared variables this should be taken into account. However, application of the "rendez-vous" mechanism for inter-process communication is recommended. Example:

```ada
    task A;
    task B is
        entry CALL;
    end B;

    task body A is
        begin
            ...........
            B.CALL    <--- rendez-vous ---> accept CALL
            ...........
        end A;
    task body B is
        begin
            ...........
        end B;
```

The execution of the accept statement results in blocking of B when A did not yet reach the *entry call* B.CALL. A is blocked if B did not yet reach the accept statement. B can have a rendez-vous with each other task that executes the entry call B.CALL. During the rendez-vous data can be exchanged. In that case the types of the parameters have to be specified in the *entry* statement, while in the accept statement and the entry call the actual parameters should be specified, e.g.:

162
task ADD is
  entry CALL( I1 : in INTEGER; I2 : out INTEGER; I3 : in out INTEGER );
end ADD;

task body ADD is
  J1,J2,J3 : INTEGER;
begin
  accept CALL( J1 : in INTEGER; J2 : out INTEGER; J3 : in INTEGER ) do
    J2 := J1 + J3;
  end CALL
end ADD;

Remark that in the example the new value of J2 is calculated during the rendez-vous. The entry call ADD.CALL(N1,N2,N3) thus results in a value of N2 that is equal to N1 + N3 and a synchronization of both tasks.

Execution of an accept statement results in blocking a task until a rendez-vous has occurred, unless the accept statement is part of a select statement, e.g.:

select
  accept MESS1;
or
  accept MESS2;
or
  accept MESS3;
end select;

Only one of the accept statements is executed: the first accept statement of which execution is possible. In stead of an accept statement also a delay statement or a terminate statement can be used. With the delay statement the maximum waiting time can be specified. The first statement after the select statement is executed if no rendez-vous has occurred within the interval specified. Use of the terminate statement results in termination of the task when a rendez-vous is not immediately possible.

An else clause can be added to the select statement, e.g.:

select
  accept MESS1;
or
  accept MESS2;
or
  accept MESS3;
else
  ........
end select;

The code after else is executed if a rendez-vous is not immediately possible. The delay statement, the terminate statement and the else clause cannot be used simultaneously in a select statement.
The select statement can be used with guards, e.g.:

```
select
  when CONDITION1 =>
    accept MESS1;
or
  when CONDITION2 =>
    accept MESS2;
or
  accept MESS3;
else
  ........
end select;
```

The programmer can e.g. use the following attributes in the tests:

- **T'CALLABLE**: Boolean variable that is FALSE when the task T is completed, terminated or abnormal
- **T'TERMINATED**: Boolean variable that is TRUE when the task T is terminated
- **E'COUNT**: Integer variable, that the number of entry calls, queued for the entry E contains

The `select` statement can also be used together with entry calls, e.g.:

```
select
  AAA.MESS1
or
  BBB.MESS2
else
  null;
end select;
```

The first entry call is executed and interrupted if a rendez-vous is not immediately possible, after which the second entry call is executed and also interrupted if a rendez-vous is again not immediately possible. The `else` clause has as effect that the task is not blocked when a rendez-vous is not immediately possible. The `select` statement for entry calls can also be used without an `else` clause. In stead of the `else` clause it is possible to use a `delay` statement, which can be used to generate a time-out condition. (This time-out is a time-out on the starting of the rendez-vous and not on the duration of it, for which Ada has no facility). The use of guards in a `select` statement for entry calls is not possible.

Entry calls for one `accept` statement are queued in a FIFO (First In First Out) queue.

The priority of a task can be specified with a `pragma` (which is a command to the compiler), e.g.:

```
pragma PRIORITY(5);
```
The effect of different priorities is implementation dependent. A task with a low priority will in general be blocked when a task with a higher priority becomes executable when both tasks run on the same processor.

A task can be terminated (i) with the terminate statement, after which the task is in the terminated state, (ii) after execution of the last end statement when all tasks for which the task is functioning as parent have finished execution, after which the task is in the completed state, (iii) by another task with the abort TASK_NAME statement, after which the task is in the abnormal state.

"Elaboration" of the declaration of a task creates the task, but does not yet activate it. This occurs after execution of the first begin statement after the declaration of the task. However, it is possible to delay the activation by making use of a special task type ("access type"), which has to be activated explicitly (with new), e.g.:

```
activation of the task :     CCC: A := new T;
declaration of A :
    type A is access T;
declaration of T :
    task type T is
        entry AAA;
        entry BBB;
    end T;
```

The entry points of the task have now the names CCC.AAA and CCC.BBB; the name of the task itself is T.all. The parent of such a task is the task in which the task has been declared as such.

Interrupts can be transformed into entry calls.

When using Ada in an MIMD environment (14), the different tasks will have to be distributed over the available processors. Specification of this distribution is not a part of Ada. The underlying operating system can in principle take care of simulating shared variables, when no common memory is present.

At present, the use of Ada in MIMD systems is not yet a matter of routine, however, the Ada tasking model provides a powerful framework for programming this type of systems.

3.2. Occam

The occam language (15) makes it possible to define interacting concurrent processes. For the communication between processes message passing is used. Occam was developed by INMOS and is geared towards application on the transputers (7) manufactured by INMOS.
An occam process consists of a sequence of actions, where an action is an assignment (result: change of the value of a variable), input or output. The actions themselves are in occam terminology again (primitive) processes. Processes communicate with each other via one-way channels (a transputer has four ports, each consisting of an input and an output channel). The communication is synchronous (blocking).

Primitive processes are:

:= assignment
?
input, e.g. link1 ? x;y performs two inputs from the channel with name link1 and places the results in x and y
!
output, e.g. link2 ! z outputs the value of z to the channel with name link2

The following "constructs" are available:

SEQ a sequential process that executes its component processes one after an other
PAR a parallel process that executes its component processes in parallel
ALT an alternative process is used to accept the first message available from a number of channels (analogous to Ada select statement)

IF, WHILE, FOR analogous to conditional and loop statements in languages like Pascal.

It is important to note that occam at all levels works with processes, so that the FOR construct in combination with the PAR construct can be used to produce a number of concurrent processes.

A few examples:

a.

CHAN inter:
PAR
WHILE TRUE
VAR x:
SEQ
in ? x
inter ! x + x
WHILE TRUE
VAR x:
SEQ
inter ? x
out ! x - 2

In this example there are two processes that execute concurrently. The first one inputs x from channel
in and outputs \( x + x \); the second one takes the output of the first process, subtracts 2 and outputs the result on channel out. Both processes consist of component processes that are executed sequentially.

b. An \( n \) stage FIFO, with for each stage a component process, and which uses a vector of channels:

\[
\begin{align*}
& \text{CHAN inout}[n+1]: \\
& \text{PAR } i = 0 \text{ FOR } n \\
& \quad \text{WHILE TRUE} \\
& \quad \text{var } x: \\
& \quad \text{SEQ} \\
& \quad \quad \text{inout}[i] \ ? \ x \\
& \quad \quad \text{inout}[i+1] \ ! \ x
\end{align*}
\]

c. Processes can be given names, e.g.:

\[
\begin{align*}
& \text{PROC buf (CHAN c1,c2)} \\
& \quad \text{WHILE TRUE} \\
& \quad \text{VAR } x,y: \\
& \quad \text{SEQ} \\
& \quad \quad c1 \ ? \ x \ ; \ y \\
& \quad \quad c2 \ ! \ x \ ; \ y \\
& \quad \text{CHAN inter1,inter2:} \\
& \quad \text{PAR} \\
& \quad \quad \text{buf(in,inter1)} \\
& \quad \quad \text{buf(inter1,inter2)} \\
& \quad \quad \text{buf(inter2,out)}
\end{align*}
\]

In this example the three parallel processes act as a buffer with room for three pairs of data items, which are passed to the buffer via channel in, and are taken out of the buffer via channel out.

Occam provides support for associating the components of an occam program with physical resources. The parallel processes created by the PLACED PAR construct can be assigned to physical processors. Channels can be associated with physical links.

Processes running on the same processor can be given different priorities using PRI PAR. This is the PAR construct, but all component processes should run on the same processor. The processes all get different priorities, such that the first component process has the highest priority and the last component process the lowest priority. Only the highest priority process is executed, unless it is blocked because it is e.g. waiting for incoming data on a channel. If several processes of the same priority are executable, then round-robin preemptive scheduling is applied. It is also possible to use PRI ALT. This is the ALT construct, but with the first component process having a higher priority than the following process, etc. The ALT construct selects one of the alternative processes for execution as soon as such a process is ready for execution. When a PRI ALT is used and a higher priority process becomes executable, while a lower priority process already had been selected, then this higher priority process takes over.

Interrupts can be associated with an occam channel.
One of the nice things of occam is the close connection with the transputer, of which the hardware has been optimized for the use of this language, i.e. process switching times are short (< 3 µs), there is hardware support for time-slice scheduling, the inter-transputer links offer the facilities that the occam channels offer. For the transputer also conventional languages (C, PASCAL, FORTRAN) are or will become available, which can be extended with features of occam.

3.3. The DRM system

The Distributed Real-time Multiprocessor (DRM) system is a product of Philips Nederland (16). It is a distributed real-time operating system for a host-target system, consisting of a mini- or microcomputer as host (VAX of Digital Equipment Corporation with VMS as operating system, or Philips 68000 based UNIX system, resp.) and 68000 family CPU's in a VME environment as target processors. For the communication between the target processors the VME-bus or RS-232-C connections are used; communication with the host is done via an RS-232-C connection. In a later stage use of Ethernet will be possible. It is possible to add drivers to DRM for arbitrary types of communication links.

The CPU modules presently supported have a 68000 CPU with or without a memory management unit (MMU), dual-ported memory (DPM), a VME-bus interrupter and two RS-232-C ports.

A single processing element is called a HAMA = HArware MAchine. One or several SOMA's = SOftware MACHines can be executing on a HAMA. A job consists of one or several SOMA's, which can be executing on one or several HAMA's. Individual SOMA's cannot be started or stopped; only jobs can be started or stopped. If CPU modules with MMU's are used, the MMU's take care of protection of the code and data segments of the SOMA's. Each SOMA can contain one or several processes. Protection within a SOMA is not provided.

For the communication between processes shared variables (only within a SOMA) or mailboxes can be used. Synchronisation between processes can be achieved using semaphores (only within a SOMA) or using mailboxes, because an attempt to read from an empty mailbox blocks the process until a message in the mailbox has arrived. Mailboxes are assigned to processes. Each process can only fetch messages from its own mailboxes, but can send to all other mailboxes. Each mailbox has a name, consisting of the name of the SOMA to which the process belongs and an identifier of the mailbox itself, e.g. SOMA1.MAIL1 is a mailbox belonging to SOMA1 with identifier MAIL1. The operating system takes care of delivering the messages at the correct destinations. The information needed is stored in the distribution file on the host computer. When a different distribution of SOMA's over the HAMA's available is desired, then only modification of this file and again starting of the job is necessary. The operating system makes use of a nameserver in the case that the destination mailbox belongs to an other job than the sender of the message (not yet implemented).
The size and the number of slots in a mailbox and the type of the information to be received in a mailbox have to be specified, and can be adjusted within limits imposed by the hardware and the size of the operating system and the user software. The communication is blocking, but, while in Ada and occam the sending process is blocked until the receiving process has actually consumed the message, here the sending process is blocked until the message has been delivered in the destination mailbox. The receiving process is blocked when it tries to consume a message from a mailbox, while no message is available, until a message has arrived. Inspection of the number of messages available in the mailbox is not possible.

Communication via the VME-bus is done by DRM by writing the data to be sent to a given CPU in the DPM of that CPU. The CPU sending the data causes an interrupt for the receiving CPU by using its VME-bus interrupter. Of the 7 interrupt request lines of the VME-bus, 6 are used for inter-processor communication. Each CPU should only be sensitive for one of those lines, i.e. at maximum 6 CPU’s per crate can be used.

Only jobs can be started or stopped, while the scheduling of different processes present on the same HAMA is done using time-slicing.

For each process a priority and a privilege have to be specified. The priority and the length of the intervals, during which processes have been blocked (when not waiting for the arrival of a message), determine which process will be scheduled. The privilege needs to be high enough to perform a number of privileged actions, like starting / stopping a job and getting access to "hard" addresses (for memory-mapped I/O).

Interrupts can be handled by user processes.

Every process has access to the file system of the host computer with the normal file I/O routines: the filenames used should have a prefix, indicating that the file is localized on the host.

Processors can be connected with each other via several communication links. With each link a cost factor is associated. The correct functioning of the links is checked regularly. The operating system makes use of the link with the lowest cost factor available and switches to an other link, if available, if the link used goes "down". If no link is available, causing that parts of a job cannot reach each other any longer, then the job is "severed" and is stopped completely.

The operating system should be "built" by the user according to the hardware configuration used. After this is done the system can be "booted" on the hardware: one CPU gets the operating system code from the host and stores it in its own memory or sends it to other CPU's in the system.

Programming languages that can be used with DRM are C and RTL/2. In a later stage it will also be possible to use FORTRAN.
3.4. FADOS

FADOS (17, 12) is a real-time distributed operating system, developed at NIKHEF-H and at the Computer Science Department (FVI) of the University of Amsterdam. It is used in a host-target environment, consisting of a system with UNIX as operating system (at present: SUN workstation, Microproject LOVME and VAX family of computers), or an Apple Macintosh with MACVEE extension (18) as host and 68000-family of CPU's as target processors (presently supported processing elements: FAMP (68000) (19), Data-Sud CPUA1 (68010), ROBCON VMECPU020 (68020)). The host computer and one of the target processors are connected to each other via shared memory or via Ethernet. Communication between the target processors proceeds via shared memory, using message passing. It is possible to add drivers to FADOS for arbitrary types of communication links.

The communication between processes proceeds similar to the Ada rendez-vous, using blocking send and receive calls. A process receiving a message has to provide an answer to the sender via the reply call. A process is addressed via its process identifier and via the name specified in the receive call towards which the message is directed.

Processes that should communicate to each other have to be loaded as one group at the same time. They can be stopped individually. It is possible to choose between preemptive or non-preemptive scheduling.

Interrupts can be transformed into messages after a call to the system primitive available for that purpose. The process receiving such a message is scheduled with the highest priority.

Every process has access to the file system of the host computer with the normal file I/O routines.

Information on the hardware configuration is stored in a configuration file, which is used when booting the operating system. Rebuilding of the system is not necessary when the configuration is changed.

Programming languages that can be used with FADOS are C, Pascal and FORTRAN when a host with UNIX as operating system is used and C when an Apple Macintosh acts as host. It is planned to support FORTRAN also for the Macintosh version.

4. Discussion

Two important types of application of MIMD systems are:

1 - computing-intensive problems, for which a single-processor system does not offer sufficient computing power or is too expensive. Example: image processing for robot vision.

170
real-time systems with many interfaces that are serviced most easily by local processors, and fault-tolerant systems. In this type of systems the computing power needed is often less important than ease of programming, flexibility and protection against software and hardware faults. Example: control system for a chemical plant.

Also in high-energy physics we see these two types of application: an example of the first category being programmable trigger processors, an example of the second category being data-acquisition systems.

For the first type of application the communication overhead in combination with the amount of communication necessary is an important factor, e.g. it makes no sense when it takes a millisecond to send a message from one processor to an other processor, which is typical for the time needed when using 68000 processors and when using shared memory, if the system should take a trigger decision in about the same time. In this case one has to resort to either communication via shared variables using shared memory or to use special hardware. The transputer in combination with occam forms a nice example of what can be done in this field. However, communication may still be a bottle-neck. In that case SIMD processor arrays probably offer a promising alternative.

For the second type of application communication overheads are not so overwhelmingly important. In high-energy physics experiments often high data transport speeds are required, but there DMA devices for transporting large blocks of data may be used. When we think of the control system for a chemical plant we are dealing with a large number of different type of sensors and actuators, that can be involved in monitoring and controlling many different processes, which can be dependent of each other. It can be quite dangerous if something goes wrong with the controlling functions of the system, so strict requirements have to be applied to the reliability of the hardware as well as of the software. Furthermore the complexity of this type of applications is generally large, requiring an effort of many man-years to construct the necessary software. The development of the Ada language, as well as of the DRM system, was motivated by this type of application.

References


OPTICAL STORAGE

G.J. Verhaart

Optical Storage International, Eindhoven, The Netherlands

ABSTRACT

In the lecture, a short introduction to optical recording has been given. Both basic theory and practical implementation are covered.

1. INTRODUCTION

Optical storage technology employs the small wavelength of light to store and read out information, with a density corresponding to that wavelength, on the surface of a so called optical disc. Despite of the small dimensions of the information structures, a large working distance between disc and recording head is possible.

In section 2 the basic principles of optical recording and read-out and the construction of optical discs, especially those for data recording, are covered.

In section 3 the various optical disc systems are covered, with special attention to the encoding and organization of data in data storage systems.


2. OPTICAL READ-OUT

2.1 Basic principles of optical read-out

Fig. 1 shows a schematic view of the light path in an optical read-out system. Light emitted by a laser (either a gas laser or a light pen consisting of solid state laser and collimating optics) is focussed on a reflecting information layer. Seen from the lens, the information layer is on the rear side of the disc. Therefore, the light beam will have a large cross sectional area at the external surface of the disc, rendering the system relatively insensitive to damage and dust. The information layer is protected against damage from the rear either by assembling two discs to a double sided structure or by applying a protective coating. The light is scattered by the information structures on the disc and partly returns through the objective lens. By the beam splitting mirror it reaches the detector. In its most simple form
L<sub>1</sub>: Objective lens  
L<sub>2</sub>: Collimating lens  
S: Beam splitter  
D: Detector  
θ: Numerical operture = sin θ

Fig. 1: Schematic view of the light path in an optical read-out system.

This beam splitter is a half silvered mirror. More commonly, however, a quarter wave retarder is inserted between beam splitter and objective. This quarter wave plate changes the state of polarization of the laser beam from linear to circular. After passing through the retarder again, the reflected light will have a linear polarization at right angles to the original polarization. By employing a polarizing beam splitter, reflected light will reach the detector without losses.

The width of the energy distribution in the read-out spot is on the order of magnitude Λ/NA

where Λ: wavelength of the laser  
NA: numerical caperture, being defined as the size of the half vertical angle of the focussed beam.  
(In optical recording NA is typical 0.4 or 0.5, the wavelength of a solid state laser is about 800 nm).

The exact intensity distribution in the focal plane is determined by the intensity distribution in the lens pupil. A homogeneously filled lens will yield the well known Airy pattern, a series of concentric light and dark rings, the diameter of the first dark ring (enclosing most of the energy) being 1.22 Λ/NA.
To reach these diffraction limited (i.e. spot shape determined by $\lambda/NA$) conditions, high quality optics are required, either a three or four lens objective or a single lens with aspherical surfaces. Also, the disc is an integral part of the design, a diffraction limited spot will only be formed on a disc of specified thickness.

Because of the small dimensions of the information carrying structures, active spot positioning is necessary.
- The objective must be movable in the vertical direction to adjust the position of the focal plane to movements of the information surface due to unflatness and vibrations of the disc.
- In the radical direction, both a coarse positioning system to access a certain track, and a fine positioning mechanism to follow irregularities in the track (due to e.g. defects or eccentricity). For coarse access, various types of carriages or swinging arms are used. A carriage may either contain all optics or only the objective and a 45° mirror. In the latter case, laser and detectors are fixed to the base plate. Fine positioning is possible either by moving the objective in a plane perpendicular to the beam's axis or by means of a pivoting mirror.

![Diagram](image)

**Fig. 2** Radial positioning of the spot

Coarse positioning:
a) the entire optics assembly, as shown in fig. 1, may be fixed to a carriage or to a rotating arm
b) laser and detector may be fixed to the baseplate, with only objective and a 45° mirror on a carriage, as shown above.

Fine positioning:
c) in case construction b) above, the 45° mirror may be rotatable, as shown in the figure
d) in both constructions a) and b), the objective may be movable perpendicular the beam axis

2.2 Theory of read-out

The reflected beam contains all information to be read from the optical disc. This means that the structure of the
disc must be such that variations in either phase or amplitude of the reflected light can be induced.

First phase structures (i.e. relief in the disc) will be discussed. This type read-out structures is very well suited to mass replication of information by embossing relief structures to discs.

An optical disc can be described as a two dimensional diffraction grating. In the radial direction the grating is purely periodical because of the fixed track pitch, parallel to the tracks it is pseudo periodical because of the regular sequence of information carrying pits.

According to the well-known law of optics, light is scattered into clearly defined directions.

\[ \sin \theta_i + \sin \theta_r = \frac{m \lambda}{p} \]

Where \( \theta_i \), \( \theta_r \) are the angles of incidence and reflection, respectively, \( p \) is the grating period and \( m \) is the order number.

In an optical read-out device, the incident beam is conical, resulting in conical scattered beams, that are partly imaged on the detector, as illustrated in fig. 3.

**Fig. 3**
Image of the first orders on the detector for a spatial frequency 
\[ \frac{NA}{\lambda} < \frac{1}{p} < \frac{2NA}{\lambda} \]
p: spatial period
x: position of the light spot with respect to the grating
A₀, A₁: amplitudes of zeroth and first orders
ψ: phase of first order with respect to zeroth order
Iₛ: integrated signal on the defects
I₅: difference signal of both detector halves

It is clear from this description that no information can be detected when the first orders are completely outside the lens pupil. In that case the spatial cut-off frequency
\[ \frac{1}{p} = \frac{2NA}{\lambda} \]
is exceeded. For frequencies below the cut-off frequency, the modulation transfer is larger when the frequency becomes lower, because the fraction of the first orders that returns into the lens pupil is larger.

2.3 Tracking and focussing systems

Track to track distances typically being only about 1.6 μm, an active tracking system is essential. For sensing the position of the spot with respect to track, there are various possibilities.

In fig. 4, the signals Iₛ and I₅, as defined in fig. 3, are depicted. The sum signal Iₛ, has minima at the position of information structures and therefore is the normal channel to read out information.

Fig. 4 Iₛ and I₅ (see fig. 3) as a function of spot position
However, as such it is not suitable for tracking, because it is symmetrical around the track position (assume fig. 4 to be a radial cross section). The difference signal, $I_d$, on the other hand, being antisymmetrical with respect to track position, can be used as a tracking signal. This method, although being very simple, also has some disadvantages. First, as is seen in fig. 3, $I_s$ and $I_d$ contain the factors $\cos \Psi$ and $\sin \Psi$ respectively, which clearly cannot be at maximum simultaneously. $\Psi$ being depended on geometry, the use of $I_d$ as a tracking signal requires a careful choice of geometry (to simplify things, narrow and deep structures favour $I_s$, shallow and wide structures favour $I_d$).

Secondly, $I_d$ is liable to DC-offsets when, by misalignment, the spot is slightly displaced on the detector. Various tracking systems, employing $I_s$ have been developed. As indicated in fig. 5, these methods either employ multiple spots, a spot that is undulating around the track centre line, or features in the disc that are displaced with respect to the track centre line.

It is easily seen that, by processing the signals in a proper way, each of these methods will provide a tracking error signal. However, compared to the tracking error signal by $I_d$, complications in optics (multiple spots), mechanics (undulating spot) or disc format (tracking flags) are involved.

In actual systems, sometimes $I_d$ provides the tracking error signal, and one of the other methods is used to compensate DC offsets.

---

Fig. 5: Various methods to generate a tracking error signal.

---

To maintain proper focus is even more important than tracking, the depth of focus being one and two µm, but disc unflatness being over a mm in some systems. Contrary to the tracking systems described above, most practical focusing systems do not require prerecorded information on the disc. Fig. 6 illustrates some systems.
Fig. 6: Various methods to generate a focus error signal.

In the knife edge method, the reflected beam is focussed onto a split detector and a knife-edge is placed in the beam. It will be clear from geometrical optics that this method will generate a proper focus. A more balanced system is obtained when the knife-edge is replaced by a prism. A second method requires the introduction of a cylinder lens in the reflected beam. The system is aligned in such a way that the circular spot midway between the two focal lines represents proper focus.

2.4 User recordable discs

Above, the read-out of discs with phase (relief) structures, typically made by some process of mass replication of identical information and employing silver or aluminium as a reflecting layer, has been discussed. In this section, discs that can be recorded by the user will be described. Those discs generally also contain embossed information for servo (tracking, synchronization, addressing) purposes. A common characteristic of recordable discs is a sensitive layer that will reflect light (for read-out) but also absorb. At normal read powers this layer is stable, but at elevated powers the heating of the layer will cause changes in its optical characteristics (in most cases the reflectivity). Those changes can be read out directly. No chemical processing or whatsoever is required.

In media, presently available, the changes are irreversible: the so called write-once systems. Fig. 7 illustrates some typical write-once layers. In ablative writing the sen-
sitive layer (often a Te alloy) is molten and the material contracts into a rim, leaving a hole (=low reflectivity) in the mirror. Bubble forming media employ a metal-polymer bi-layer. Upon heating, the polymer decomposes and deforms the metallic layer into a bubble. A trilayer structure consists of a thin sensitive layer on top of a dielectric spacer and a total reflector (e.g. aluminium), forming an anti-reflective structure (some residual reflection is, of course, necessary to maintain focus). Almost all incident light is absorbed in the top layer, enabling hole forming at relatively low powers. A bright mark results. A disadvantage if this structure is that read-out must be air incident, requiring special protective structures.

Sensitive layers that allow reversible writing are now in development stage. Two examples are described in fig. 8

Phase change-recording uses the different reflectivity of crystalline and amorphous states of certain materials. Heating above the melting point and rapid cooling (short, high power pulse) will quench the material in its amorphous state. Annealing by heating the material just below its melting point (short, low power pulse) will cause erasure. The evident problem in developing suitable layers is that erasing must be possible in less than a μs, but the written effect, nevertheless, must be stable over years at ambient temperatures.

In magneto-optic recording, the magnetization of a suitable sensitive layer may be changed by heating it (with a laser pulse) above the Curie temperature in an external field. Read-out, requiring no external field, is accomplished by the Kerr effect (i.e. rotation of the polarization of the incident light). The obvious advantage over high density magnetic recording is that an optical read-out system with a
Fig. 8: Two principles of erasable recording.

large working distance and a bulk magnetic field is used as opposed to a magnetic head with very low flying height.

3. SYSTEMS

First read-only systems, like video and CD-digital audio, employing relatively cheap media that can be replicated in large quantities will be described. Subsequently user recordable systems will be discussed.

3.1 Laservision (VLP)

An analogue video system, based on double sided Ø 30 cm plastic (generally injection moulded) discs. CLV (constant linear velocity) discs maintain the same high density throughout the disc and offer one hour of playing time per side. CAV (Constant angular velocity) discs have a reduced playing time, about half an hour per side.

Because exactly one video picture has been recorded per revolution, fast play back, slow motion, still picture and tracks are easily accomplished in CAV type discs. Also, 50,000 still pictures can be stored in this mode. The VLP was the first optical disc system to be introduced to the market, but, in its original consumer application, has never been very successful, a.o. because of competition by VCR and by an apparent lack of appraisal of its far superior picture quality. At present, interactive (CAV) video discs, often connected to a computer, are finding their way into various professional applications.
3.2 CD-digital audio/CD-ROM

The audio system is based on single sided plastic 12 cm discs. Playback is in the CLV mode. Both audio channels are sampled with 44.1 KHz sampling frequency and 16 bit quantisation. On a disc with a one hour playing time more than 600M bytes of digital information are recorded. Because of this large capacity, the CD disc is also a useful medium for the distribution of software or data bases. In that case, it is called CD-ROM.

The data, recorded to the disc, are encoded in a certain modulation system that must meet the following requirements:
- highest possible density within the limitations of optical read-out
- self clocking which requires a minimum number of transitions per second
- no low frequency components, to prevent noise in servo systems

In CD, EFM (8→14 modulation) has been chosen. Eight data bits are translated into fourteen channel bits. The first two requirements are met because this is a runlength limited code, each pit or space between pits has a minimum length of eleven channel bits.

In the replication process or by later mishandling a disc may be damaged. To prevent the audio channel of being affected
- redundant information is added
- information is interleaved to reduce the influence of local defects
- when, even with use of the redundant information, samples can not be reconstructed, a value is assigned by interpolation between the adjacent samples

The combination of these techniques allows for full correction of errors with a length of 4,000 databits (2.5 mm of track) and correction by interpolation of 12,000 data bits (7 mm of track). For CD-ROM, interpolation is not an option. Here, a second layer of error correcting code is added.

3.3 Recordable systems

The read-only systems discussed above are both industry standards. On the contrary, optical disc systems for data recording have not been standardized. Various, widely different, systems are offered.

At present, write once systems are available, mainly the systems using 30 cm discs made out of glass or plastic and storing about 1 G byte/side. A large number of companies is developing or already marketing systems based on 130 mm diameter discs, generally employing plastic discs. Contrary to the 30 cm systems, there is a reasonable chance that a
world standard will emerge. With respect to the field of application write once disc systems must not be compared to magnetic disc systems. Most systems are being used for archival purposes, replacing microfiche or tape. Optical discs are especially suited to these applications because of their removability and long life (over ten years), combined with rapid access to the data (often jukebox systems are applied).

Although there are large differences among the systems offered by different manufacturers, there are some common characteristics:

- all systems work in a CAV made to enable fast random access
- because tracks of about 1,5 μm spacing must be followed with a relatively simple drive and on a removable disc, active tracking is unavoidable. Almost all systems employ prerecorded information, embossed by the same techniques as used in the production of read only discs, for tracking. The structure is different, however, from read only discs, where the sequence of information carrying pits forms a track. Two approaches are possible:
  * some systems have a continuous, shallow groove, which yields a good tracking error signal in the radial difference signal (I_d, see section on tracking). User data is superimposed upon this groove.
  * other systems employ the so called sampled servo method. Here the (imaginary) track is subdivided into parts reserved for user data and parts containing servo information. For tracking normally tracking flags as shown in fig. 5 are used.
- discs contain prerecorded track and sector numbers. Normally, discs are divided in sectors containing 1024 or 512 user bytes. Error correction systems and interleaving of information, as described for CD, are employed.
- data is normally encoded using some modulation systems adapted to the requirements of the optical channel.

As described in the section on sensitive layers, all optical discs can be read immediately after writing. This enables the system to check the written information. The various types of optical drives either read-back during write or read in a subsequent revolution (the latter method of course halves the data rate, so the user may decide not to check at all). If, upon checking, written information is found to be suspect, (the definition of suspect depends on the opinions of the system designer, one system will reject sectors with only a few missing bytes, other systems will reject only sectors exceeding the correction possibilities of the error correction system) it can be rewritten in another sector. This sector can either be the next sector, which will hardly affect transfer rate but will mix up logical and physical sector numbers, or a sector in a special reallocation area.
In the table, capacity of the read only system CD-(ROM), of the present 30 cm write once discs (the OSI system has been chosen as an example, other systems are comparable) and the more recent 130 mm discs are compared. It is seen that read only systems allow for a higher density. However, density of recordable systems is increasing and a capacity of 2 G byte per side will be possible in a next generation of 30 cm discs.

<table>
<thead>
<tr>
<th>READ ONLY</th>
<th>CD ROM</th>
<th>OSI 30 cm disc</th>
<th>OSI/SONY standard proposal for 130 mm discs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACK PITCH (µM)</td>
<td>1.6</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>BITLENGTH, RAW (µM) inner radius</td>
<td>.67</td>
<td>1.84</td>
<td>0.95</td>
</tr>
<tr>
<td>BITS/MM² (RAW)</td>
<td>10⁶</td>
<td>.35 10⁶</td>
<td>.70 10⁶</td>
</tr>
<tr>
<td>OVERHEAD</td>
<td>.27</td>
<td>.37</td>
<td>.34</td>
</tr>
<tr>
<td>USER BITS/MM²</td>
<td>0.7 10⁶</td>
<td>0.6 10⁶</td>
<td>.46 10⁶</td>
</tr>
<tr>
<td>CAPACITY OF ONE SIDE (USER)</td>
<td>550 MBYTE</td>
<td>1 GBYTE</td>
<td>320 MBYTE</td>
</tr>
<tr>
<td></td>
<td>(CLV, 12 CM)</td>
<td>(CAV, 30 CM)</td>
<td>(CAV, 13 CM)</td>
</tr>
</tbody>
</table>
1. Introduction

A Graphics System may be seen as a computer-aided environment supporting the application program. The basic elements associated with it are:

- Operator (User)
- Graphics Support System (Services)
- Other User Interface Support System (Services)
- Application Functions
- Generic Action Routines
- Data Base

The kernel for the communication between the different parts of such a system is composed of the different data items associated with the communication elements.

From the point of view of the user operating such a graphics support system, a simple functional model may be given showing the functional decomposition of the model. The Graphics System is then based on the following basic functional components:

- User (Operator of the Computer-Aided Environment based on a Graphics Support System)
- Graphics
  - Graphics Viewing
  - Request Processing
  - Graphics Metafiles
- Model
  - Model Metafiles (Design Data)
  - Model Data Management
- Application
  - Control
  - Heuristics

This model implies a clean separation between application system, application control, graphics support system and model data management functions. This functionality again is based on the transfer of the associated items stored in the corresponding metafiles (graphics, model).

2. Data Interfaces

The model shown gives the basic functionality of a Graphics System as a computer-aided environment supporting the application program. The model given in Fig. 1 shows the different data interfaces in such a system for which standards have been developed or are in the process of being developed; these are:

- GKS and GKS Language Bindings; this is also the interface required for other graphics functional standards under discussion (e.g. PHIGS)

- the European Videotex Interface CEPT Geometric Encoding (which is a subset of the ECMA Graphics Data Syntax (GDS))

- Computer Graphics Interface (CGI) formerly known as Virtual Device Interface (VDI)

- Computer Graphics Metafile (CGM) formerly known as Virtual Device Metafile (VDM)

- the US Videotext Interface, the North American Presentation Layer Protocol Syntax (NAPLPS)

- Product Definition Data Interfaces like the Initial Graphics Exchange Specification (IGES) for the exchange of product data in connection with the graphic model, the VDAFS Sculptured Surface Interface developed by the German Automobile Manufacturers, SET (Aerospatiale, France) or XBF-2 (CAM-I); other examples are the German work on Transport and Archiving of Product Definition Data (short TAP) and the related international work on STEP, the planned International Standard for Exchange of Product Definition Data Models.

- Interfaces for the integration of graphics in documents, e.g. the integration of graphics metafiles into documents defined in Standard General Markup Language (SGML) or Office Document Architecture / Office Document Interchange Format (ODA/ODIF).

- Interfaces for Presentation Graphics on the top of graphics standards like GKS.

The coherent and efficient integration of all these standards and developments into a highly functional Graphics Support System is one of the technical goals for the future in the area of Computer Graphics System Design.

Two activities related to GKS will be of specific impact in this development:

- the VLSI support of GKS implementations, and

- the design and implementation of GKS based applications for open communication systems (e.g. LANs and WANs).
3. Standards for Graphics Programming

In the history of Graphics standards development several milestones are to be considered:

- Formation of the Graphics Standards Planning Committee (GSPC) in 1974 by ACM SIGGRAPH /1/;

- Formation of the Committee for the development of Computer Graphics Standards by the German Standardisation Body (DIN - Deutsches Institut für Normung) in 1975 /2,3/;

- The IFIP WG5.2 Workshop SEILLAC I (Methodology in Computer Graphics) organized by R. Guedj in France in 1976 /4/.

Today's scenarium of these standards and projects is shown in Fig. 2. This figure shows the distribution of graphics standards based on their dimensionality (2D and 3D), level of picture structuring (segments and hierarchies), and specification type (Functional Specifications, Language Bindings, Data Encodings):

- The Graphical Kernel System GKS is an already well established and publicised standard (ISO 7942) /2,3,5,6,7/. GKS will not be further discussed here, since it is well documented and discussed elsewhere /2,5/ in substantial detail. GKS is a 2D standard with one level of picture structuring based on segments.

- GKS-3D /8/ provides the application program with the following capabilities:
  - The definition and the display of 3D graphical primitives;
  - Mechanisms to control viewing transformations;
  - Mechanisms to control the appearance of primitives including optional support for hidden line and/or hidden surface elimination, excluding light source shading and shadow computation;
  - Mechanisms to obtain 3D input.

Fig. 3 shows the GKS-3D viewing pipeline. Existing 2D GKS applications should run without modification on systems incorporating GKS-3D, since no changes are made to the existing 2D functions. GKS-3D by now has reached ISO 2nd DP status being registered as ISO DIS 8805 early 1986.

- PHIGS (Programmer's Hierarchical Interactive Graphics Standard) /9/ includes in its functionality three dimensional output primitives and transformations and hierarchical segments, called structures. It has dynamic control over the visual appearance of attributes of primitives within a structure. One goal of PHIGS is to support the most powerful workstations becoming available. The structure of the PHIGS viewing pipeline is shown in Fig. 4.

- GKS Output Level 3 /10/ describes a set of extensions to GKS for segment hierarchy and editing. This GKS level 3 allows a segment to invoke other segments; an existing segment may be reopened for editing; elements may be inserted and deleted. The content of segment elements may be inquired. Segment networks may
be stored in metafiles. This GKS level 3 demands only 15 functions in addition to GKS. By now GKS level 3 has no official standing as a standard or standards project but is only a working document; nevertheless it shows an extremely interesting way of extending GKS to handle segment hierarchies with the appeal of needing only very few additional functions. Based on GKS-3D this approach gives (by a minimal extension) a functionality comparable to PHIGS which is at the same time fully compatible to GKS and GKS-3D.

An important issue to be addressed in this context is the migration of applications written for the different standards shown in Fig. 1. The migrations from GKS to GKS-3D or from GKS to GKS level 3 are guaranteed by the fact that the full GKS functionality is a subset of both GKS-3D as well as of GKS level 3. The migration from GKS to PHIGS or from GKS-3D to PHIGS still make some problems because of existing incompatibilities between the functionalities of the GKS Standard and the current document of PHIGS. One of the key technical issues is the attribute binding concept, where GKS binds attribute values at generation time of the primitive, whereas PHIGS does the binding only at traversal time of structures and attribute values may be inherited from the parent structure. It may be that the actual public reviewing of PHIGS will help to solve some of the existing technical problems with the migration of application programs between the GKS Standard and a planned PHIGS Standard.

The "down migration" of applications from GKS-3D (and GKS Output Level 3) to 2D GKS may be solved by using the metafile functionality.

4. Graphics Metafiles

Graphics Metafile Standards ideally define file formats for storing and transmitting pictures in a device and application independent way. In practice they define functionality as well as encodings of formats. They enable the interconnection of various graphical devices and graphics systems in a standardized way. An overview of different encoding methods is given in /11/.

GKSM is the GKS Metafile used by GKS; it is a sequential file that can be written and read by GKS /7/ and is used for long term storage and transmittal of the graphical information produced or to be read by GKS. GKS provides means not only for writing but also for reading of the GKSM. The GKS Metafile contains two dimensional pictures represented by data records (called items) generated as a result of GKS functions involved. The GKSM contains:

- file/picture header,
- end item,
- control items,
- output primitive items,
- attribute items,
- non-graphical, application dependent data (user items).

GKSM is build up of a sequence of logical variable length data records. GKS addresses GKSM like a workstation. Two workstation types are defined for the GKS Metafile:
- the GKSM output workstation and
- the GKSM input workstation.

The functionality of the GKSM is described in the GKS /7/ document. A proposal for a presentation is given in Annex E of this document.

The Computer Graphics Metafile (CGM) /12/ formerly known as VDM (Virtual Device Metafile) defines the functional behaviour (semantics) and form (syntax) or encoding of a set of elements. The following set of elements is specified:

- descriptor elements,
- control elements,
- picture descriptor elements,
- graphical elements,
- attribute elements,
- escape elements,
- external elements.

A CGM is a collection of elements from this set. The descriptor elements give the CGM interpreter sufficient information to interpret metafile elements and to make decisions concerning the resources needed for display /12/.

CGM is expected to be usable as a GKS Metafile at level 0a of GKS if the portability of a CGM for GKS applications has to be ensured.

Since CGM cannot serve as a full GKS Metafile (levels 1 and above) and there is only one format for a GKSM given as example in Annex E to IS 7942, work is now under way to provide a full GKSM which is upward compatible to CGM. This project with the working title Generalized Graphics Metafile (GGM) has been prepared by the DIN working group on computer graphics and is about to be started in the ISO arena.

5. Device and Workstation Interfaces

GKS introduced the concept of workstations. These workstations hide device characteristics from the application programmer and thereby has significantly increased device independence of graphics software. Multiple of these workstations can be concurrently active and can be controlled independently by one application. This multiple workstation concept implies an interface called the multiple workstation interface, i.e. an interface before the data and control stream is splitted up to the different workstations. A second interface is required which adresses the single individual workstation. This interface is called the Workstation Interface (WSI).

The European Computer Manufacturers Association ECMA has defined its ECMA Standard 96 /13/ called Graphical Data Syntax (GDS). ECMA GDS is a character coded binding of almost the complete GKS functionality; the only missing parts are the GKS normalisation transformation and the GKS metafile functions; all other functions including those referring to the Workstation Independent Segment Store (WISS) and the complete input functionaliy are included. Thus difference to full GKS is minimal.
The development of this standard and the adoption of the output subset by CEPT strongly supports the expectation that in the near future devices with full GKS capabilities will be available at reasonably low cost; another implication of that development is that the appearance of GKS on silicon makes the efficiency complaints about pure software implementations obsolete.

In GDS the functions are coded very efficiently taking into consideration that reduction of the amount of data to be transferred reduces cost significantly.

The WSI specifies the functional separation of the GKS Kernel and the GKS Workstation. It allows the operation of GKS in distributed systems and the exchange of information between kernel and workstation of different implementations. The distribution among different hosts does not put restrictions on the functionality defined by GKS. The WSI is only implicitly defined in the GKS document; a more specific definition is given in /16/. Within ISO a Computer Graphics Interface Standard CGI /17/ is currently under development. It is intended that this standard shall provide at least the full functionality of the Workstation Interface.

For some devices including GKS and CGM (at level 0) the device interface will coincide with the WSI; for other devices WSI capabilities not available on the device have to be simulated; such devices provide a low level device interface only and need device drivers.

A graphics device driver is that portion of the graphics system software that translates commands and data from the CGI into the form required by the particular input/output mechanism of the device. CGI is an interface which is internally closest to the physical devices and the last point where device independence can still be achieved.

6. Videotex

Graphics in videotex environment can be represented in three different ways:

- alpha-mosaic character graphics,
- scanned image facsimile mode graphics,
- geometrically encoded graphics.

This last one has been strongly influenced by and come out of the computer graphics standardization at least in Europe.

As already mentioned before CEPT, the European PTTs have adopted as their Videotex Standard Geometric Encoding /16/ the output subset of the ECMA Graphical Data Syntax (GDS) /13/. The influence of this interface cannot be estimated high enough regarding the impact this standard has on the appearance of VLSI based solutions for graphics.

The North American Presentation Layer Protocol Syntax (NAPLPS) /17/ is an older attempt to provide character encoded graphics in a videotex environment. Despite the claim NAPLPS does not encode graphics commands very efficiently, definitely not as efficiently as GDS does; compared to GDS the graphics functionality is poor; the interface is clearly closer to a rather dumb device.
Currently there are strong currents within CCITT to provide a common framework for various videotex standards. One goal of the videotex development is integration. The functionality of videotex itself currently already integrates text, graphics and facsimile. The availability of these services in larger integrated networks is another important aspect of integration.

7. Product Data Transfer Formats

The Initial Graphics Exchange Specification (IGES) /18/ is a neutral data format which serves as a communication file to transfer data between CAD/CAM systems. IGES is structured as a five section file, containing the following parts:

- Prolog Section contains user defined text header for the file;

- Global Section contains information about the system, on which the part was developed, including numeric accuracies, scale factors, units of measurement and other environmental parameters; the object data in the file is composed of individual entities;

- Directory Entry Section contains an index to all the entities in the part as well as descriptive attributes for each;

- Parameter Data Section contains the actual data defining each entity;

- Terminator Section acts as a bookkeeping record to check the number of records received and processed.

IGES files are coded in ASCII, with 80 characters per record and appear logically as card deck. The new versions of IGES allow a binary file structure to reduce the data volume. The binary file structure utilises data definitions and relationships identical to the ASCII version, but trades substantial data compression for increased processing complexity. The IGES file concept is strongly directed to the "transfer of drawings" and related information between CAD/CAM systems.

IGES includes three entity types:

- geometry (including point, line, circle, conic, parameter spline, surface of revolution etc.);

- dimensioning/annotation (includes angular dimension, centreline, label, etc.);

- structure (includes standard and user defined associations, drawing relationships, font relationships, view relationships).

IGES is the oldest of Product Data Transfer formats; it became a NBS standard and later on an ANSI standard while still at a rather immature stage. As a result of the problems experienced with IGES ongoing development of follow up standards are going on in various arenas.

192
The German Standards Organisation DIN e.g. has taken an interest in standards for graphics and CAD systems interfaces for a number of years. In the beginning of 1983, when the demand for standards in product definition data exchange began to be expressed more acutely, a new DIN working group was formed. It was named "Transfer and Archival of Product Definition Data" (briefly TAP). This working group is concerned with issues very similar to those of IGES in the data exchange between CAD systems as well as a medium or long term storage of such data.

In the area of automotive engineering in Germany, the Association of Automobile Manufacturers (VDA) has recently developed a standard format for exchanging curve and surface data, called the VDA Sculptured Surface Interface called VDAFS /19/.

The approach taken here may serve as an example of a low redundancy lean entity set solution. The interface is limited to geometry entities (and comments) and contains only the following set:

- point;
- point set;
- point vector set;
- composite curve (include parametric splines);
- parametric spline surface.

Any geometric representations not contained in this basic set are converted to the standard by the sending system and later reconverted into the local representation of the receiving system. The VDA proposal is currently under review by DIN and will become a German national standard probably in 1986. Pilot implementations exist and have been tested with good success. The VDAFS will also be submitted for international reviewing.

International work in this area is done within ISO TC184/SC4, where an International Standard for the Exchange of Product Model Data (STEP) is being developed. One source comes from the US follow up activities for IGES called Product Data Exchange Specification (PDES). Besides the DIN work another source is from the French SET /20/ (Standard d'Echange et de Transfert) from Aerospatiale.

Process automation is an area of increasing impact in the US as well as worldwide. The keyword there is MAP, the Manufacturing Automation Protocol from General Motors. This is a de facto standard for the integration of Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Computer Integrated Manufacturing (CIM). MAP gains more and more industrial acceptance worldwide and it can be expected that this also will influence STEP development too.

By the end of 1983 a new Electronic Design Interchange Format (EDIF), with similar goals to IGES, has been developed by the main semiconductor and computer companies. EDIF enables the communication of various types of electronic data among CAD/CAE tools and systems. The EDIF file consists of four kinds of information blocks:

- Status blocks contain accounting information, such as data, author's name and software level;

- Design blocks provide entry points to the EDIF file contents by indicating which cell in which library contains the top level description of the design conveyed by the file;
- Cell Definition Libraries contain all relevant design information;

- User Data blocks are catch-alls to handle information not otherwise expressible in EDIF.

The EDIF file does not concentrate as much as IGES on the drawings and related data, but more on electronic design information and corresponding data. This file is still in a very early stage of reviewing; the companies involved in this reviewing process are, among others, Daisy Systems, Mentor Graphics, Motorola, Semiconductors, Tektronix, Texas Instruments and University of California at Berkeley. There were also efforts to extend IGES for PCB design.

8. Graphics in Documents

The two major standards developments in the area of documents are the Standard General Markup Language SGML /21/ and Office Document Architecture/Office Document Interchange Format ODA/ODIF /22/. They distinguish between a logical and a layout structure, between a processible and an image form of a document. In SGML as well as in ODA/ODIF proposals are currently under discussion to integrate graphics in form of graphics metafiles. This allows to supply graphics from external sources and integrate them into a document in processible form. The document processing and producing system has to provide capabilities to handle such compound documents. This is a good example of integration of standards of different fields.

9. Presentation Graphics Layer on Top of GKS

For a Presentation Graphics Layer the following three interfaces have to be specified:

(a) Interface "GKS - Presentation Graphics Package"

- primitives
- data structures
- clusters/levels (functionality)

(b) Interface to the Environment

- operating system
- language binding
- interface to the data handling utilities
- interface to the methods handling utilities

(c) Operator Interface

- passive/interactive
- dialog
- interaction techniques
Based on such a specification, special implementations of a presentation graphics package can be realised which at the application layer differ from each other. Their efficiency is then dependent on the methods library and on the data structure handling utilities used. A Working Group of the German GKS Verein (GKS Association) is active in the development of common interfaces (a), (b) and (c) on the top of GKS, to achieve more flexible capabilities for system integration /23,24/.

10. GKS in a Network Environment

The WSI (Workstation Interface; see "Device and Workstation Interfaces") of GKS opens the possibility of using GKS in a network environment. The communication is based on the concept of a Graphic PAD (G PAD), which is an extension of the CCITT standard proposal for alphanumeric communication (PAD). Such a G PAD is a software module, with the functionality of channeling the information received either to an alphanumeric terminal, or to a given process running under the local operating system; this process implements the GKS Workstation. The GKS oriented communication protocol is based on the services supplied by the T.70 transport protocol. In Germany, several institutions are involved in designing and implementing these concepts; these activities are conducted under the sponsorship of the German Research Network DFN (Deutsches Forschungsnetz).

* * *

11. Bibliography


/6 / Duce DA, Hopgood FRA (1984) Lecture Notes of GKS Tutorial at EUTROGRAPHICS'84, Copenhagen

/7 / ISO 7942 (1985) - Information Processing - Computer Graphics - Graphical Kernel System (GKS) - Functional Description


/13/ Standard ECMA-96 (1985) Syntax of Graphical Data for Multiple Workstation Interface (GDS)

/14/ Structure of a Workstation Interface. Position Paper, DIN MI AK 5.9.4


/16/ CEPT T/CD6-1 part 2 (revised) - Videotex Presentation Layer Data Syntax - Geometric Display


/19/ DIN 66301 Normentwurf (1984) - VDAFS Format zum Austausch geometrischer Informationen (Sculptured Surface Interface) German Version

/20/ SET Specifications Rev. 1.1 (March 1984) (Standard d'Échange et de Transfert). Aerospatiale Direction Technique


/23/ Puller A, Harendra HB (March 1983) Management Graphics. CAMP'83 Tutorial, Berlin

/24/ Protokoll der Sitzung der ad-hoc-Gruppe "Präsentationsgraphik" des GKS-Vereins am 10.02.1984 in Darmstadt
Fig. 1 Functional and Data Interfaces in a Graphics System

Fig. 2 Classification of Graphics Standards and Projects
Fig. 3  Viewing Pipeline of GKS-3D

Fig. 4  PHIGS Viewing Pipeline
INCORPORATING KNOWLEDGE ENGINEERING AND COMPUTER GRAPHICS FOR EFFICIENT AND USER-FRIENDLY INTERACTIVE GRAPHICS APPLICATIONS\textsuperscript{)}

J.L. Encarnacao

Technische Hochschule Darmstadt, Darmstadt, Federal Republic of Germany

There is a wide spectrum of applications for computer graphics techniques as a tool in the problem solving process. This process is in many cases not explicitly known or given as a set of algorithms and/or application methods. Computer Graphics is then used as a tool to support the problem description and the operating interface of the knowledge-based system. The basic structure of such an application of Computer Graphics is shown in Fig. 1.

For the design of the system structure shown in Fig. 1 there are basically the following problems to be solved:

1. Object representation and knowledge representation:
   1.1 Representation of the objects
       (application primitives with their geometry and
        graphic structure);
   1.2 Representation of the knowledge about the objects
       (semantic of the application primitives;
        application context).

2. System control and operating
   (man-machine interface; graphics dialog; access to the
   methods; implementation of the inference between knowledge,
   problem description, related objects and methods needed for
   the problem solution).

3. Knowledge acquisition (to build the knowledge data base).

The architecture of such a graphics knowledge-based system is shown in Fig. 2. We see there the clean distribution between the knowledge container and the interactive graphics system for the system control and operating. Such an architecture involves a set of issues to be considered and solved when designing a graphics knowledge-based system:

(1) Implementation language and the corresponding graphics
    interface (e.g. how to interface PROLOG and GKS).

(2) Knowledge acquisition and user interface manager
    (e.g. how to interface UIM and the dialog component of
    the knowledge base).

(3) Use of graphics for the knowledge acquisition and for the
    operating of the problem solving process (e.g. how to inte-
    grate GKS and the dialog component of the knowledge-based
    part of the system).

(4) Integration of knowledge base, data base and methods base,
    with UIM-supported access.

\textsuperscript{)} Reprinted from Eurographics '85 (North Holland, Amsterdam, 1985), by kind permission of Eurographics, the European Association for Computer Graphics.
Figure 1: Graphics as a tool in the problem-solving process

Figure 2: Graphics knowledge-based system
The presentation will survey existing solutions for the problems listed above, will report on implementation experiences at the Interactive Graphics Research Group (GRIS) of the Technische Hochschule Darmstadt (PROLOG/GKS; methods-based graphics systems; graphics-based expert systems; etc.) and will discuss the trends for this area.

* * *

Bibliography

Methodenbanksysteme und wissensbasierte Techniken: Die Hauptbausteine für zukünftige CAD- und CAE-Systeme; Vortragsmanuskript; THD-GRIS (internal notes in german) Darmstadt, 1985

A GKS based programming in PROLOG; submitted for publication, 1985

As funcoes GKS na linguagem PROLOG; Universidade Nova de Lisboa; Report UNL-5/85; April 1985 (in portuguese)

Entwurf, Realisierung und Bewertung von graphisch-interaktiven Methoden-Informationssystemen; Report THD/GRIS 85-1; Darmstadt, 1985 (in german)

Models and methods for decision support systems for evaluating and choosing CAD-systems; accepted for publication in Proceedings of the IFIP 5.2-Working Conference "Design Theory for CAD", Tokyo, October 1985; to be printed by North-Holland Publ. Co., Amsterdam

Graphics programmes for knowledge-guided interaction in these Proceeding of EUROGRAPHICS'85; to be published by North-Holland Publ. Co. Amsterdam (1985/86)
MODELS AND METHODS FOR THE IMPLEMENTATION OF DECISION SUPPORT SYSTEMS AIMING AT THE EVALUATION AND SELECTION OF CAD-SYSTEMS*

J.L. Encarnacao and L.A. Messina

Technische Hochschule Darmstadt, Darmstadt, Federal Republic of Germany

Z.I. Markov

Bulgarian Academy of Sciences, Sofia, Bulgaria

1. Introduction

CAD-systems are widely used in many fields of human activities. As they are expensive and complex tools, the problems of evaluating and choosing the proper system for particular applications naturally arise.

A CAD-model is proposed in order to reduce the complexity and to automate the process of evaluation and choice of such systems. The basic idea is that if one can describe the composition of the applications in a firm, a set of necessary technical support can be found and this set can be ordered by adequacy grades.

The understanding throughout this paper of a CAD-system and its functionalities is based on /1,3,5/. The ideas presented here are the further development, the refinement and the implementations of some of the ideas presented in an early conceptual stage in /6,7/. New is the Prolog-oriented approach to the implementation of this kind of decision support systems.

Although the problem which will be treated in this paper is that of describing a procedure for evaluation and choice of CAD-systems, the general approach-model suggested here deals with a wider variety of problems, i.e. those concerned with description and handling of organizational, technical and performance/cost parameters.

2. CAD-Model Definition

This formal definition is expressed in the first order predicate calculus and its model theory. This formalization is proposed by W.Kasprzak /9/.

The positive aspect is that the transition to an implementation in form of an expert-system seems to be simple.

*) Reprinted from 'Design Theory of CAD Systems' (North Holland, Amsterdam, 1986), by kind permission of Eurographics, the European Association for Computer Graphics.
For the CAD-model we build a language \( \Omega_e \) of the class \( \Omega \).

* sorts \( /\Omega_e/ = S_{OP} \cup S_{TP} \cup S_{PCP} \),
  
  \( S_{OP} \) - The sort name for organizational parameters
  
  \( S_{TP} \) - The sort name for technical parameters
  
  \( S_{PCP} \) - The sort name for performance and cost parameters

* The correspondent set of variables \( X_e = OP \cup TP \cup PCP \)

* \( \langle CAD-model\rangle = \langle (\mathcal{M}_e, \psi), T, \Phi \rangle \)

\( \mathcal{M}_e \) - an implementation of the language \( \Omega_e \)

\( \psi = T \cup I_1 \cup I_2 \)

\( T \) - The axioms of the language \( \Omega_e \)

\( I_1 \) - Inference of the type

\( \text{formula}(OP) \rightarrow \text{formula}(TP) \)

( They describe the transition from a description in organizational parameters to a description in technical parameters )

\( I_2 \) - Inference of the type

\( \text{formula}(TP) \rightarrow \text{formula}(PCP) \)

( The transition to a description in performance and cost parameters )

\( T = R_1 \cup R_2 \cup \delta \) - the analysis methods

\( R_1 \) - so called metarule in inference form

\( \text{formula}_1 (OP) \rightarrow \text{formula}_1 (OP) \land \text{formula}_2 (OP) \)

( They describe the possibility of expansion of the given formula to a formula with additional organizational parameters )

\( R_2 \) - so called metarule, which contains statements upon the choice of the better variants

\( \text{formula}_1 (TP) \cup \text{formula}_2 (TP) \rightarrow \text{formula}_1 (TP) \)

\( \text{formula}_1 (PCP) \cup \text{formula}_2 (PCP) \rightarrow \text{formula}_1 (PCP) \)

\( \delta : t_i/x_1, \ldots, t_n/x_n \) - Substitution rules applied to inferences \( I_1, I_2, R_1, R_2 \).

\( \Phi = ( \phi_{x^*} / x^* \in X_e^* ) \) - a family of interpretations, which are distinct only in the values for the variables of \( X_e \).

This formal conception of a CAD-model still needs much consideration before it can be used effectively for the whole problem domain. For example, the set of variables \( X_e \) does not explicitly distinguish the attributes and functions associated to a subclass of \( X_e \) (see section 7). This distinction can be achieved through interpretations.
Also the set of inferences and metarules, although here defined, may suffer from incompleteness and inadequacy in trying to solve some formulations, which will possibly appear in the near future.

Since this project consists also of an implementation tool, real problems will be regularly tested and reformulations on the structure will be attained.

3. Organization Parameters

These parameters serve to describe the organization in which the CAD-system is to be used. The organization's functional behaviour and the operation of the application processes, which correlate to CAD-systems, are described by sets of parameters and rules.

The upper-level parameters are:
- Application field
  There is a hierarchical structure under an application field such as e.g. mechanical, which determines the general events and necessities of an area.
  Also, the structure of the organization ( departments, tasks and processes ) could be simply represented by a tree-structure or more detailed by a Petri-Net-structure allowing the possibility of a synchronized sequence control.
- Capital turnover
  Dynamically observed through periods of time and considering changes in technology and market strategy, this is a variable in economic methods such as e.g. Payback and Return on Investment, which serves to analyse investments and cost savings.
- Personnel and qualification
  This pictures the state of manpower available. It serves to predict the new necessities according to certain objectives to be fulfilled, as for example CAD-system introduction. It supports task distribution planning and calculation of the number and type of workstations.
- Trade union support, employees motivation and social impact
  These are important factors concerning the acquisition and introduction of new tools. Although, they are not measured adequately.
  For most of the cases, if the support is negative and/or the motivation is low, the causes are: unemployment, low decision's grade and ( maybe unconsciously from both sides ) bad training. This means that toward these and other reasons, a control mechanism should indicate where it happens and how this can be avoided; as for example by personnel replacement.
  Social impact is too complex to be treated at this point. It involves analysis of satisfaction, motivation, environment, and besides that, the data classification is not qualified for this purpose and the comprehension methods are too complex.
- Changes of communication between departments
  This can be achieved through the reconfiguration of the firm structure and additionally, if it is the case, through the integration of the network.
- Standards and documents used

According to the application field standards and catalogs of parts support the designer's work. These and the way in which they are available, are important tools. They are: norms, drawings, lists, tapes, disks, microfilms, DB's, MB's, KB's ...

- Firm planning

This should use the firm structure. Here we see the necessity of structures such as Petri-Nets in order to execute appropriately the analysis of possible paths or alternative sub-structures. The net can be used to control the many possibilities of information flux and these variations will determine the state transition occurrences. The control variables could be of a probabilistic nature, allowing the study of the occurrences through distribution functions.

- Integration to existent computer network

If integration to existent computer network is to be achieved, this existence should be technically determined. The interface for technical parameters accepts the configuration as input. This adds also some restrictions to the choice.

- Use of free personnel capacity

If personnel excess occurs, personnel and/or task transfer is necessary according to the firm structure. This is really not a problem to be so easily handled. Nevertheless, adequate training and education are in the long run the preventive solution.

- Project coordination

Tasks can be formulated in details or phases, each of which could have attached to it the personnel capacity and the computer support. Additionally both man and machine could be responsible for determined tasks. This can be easily accomplished by an adaptation to the smallest unit method (analogy to the 'smallest time method' - see section 6).

- Supplier dependence

The type of communication (as in standards and documents) and the hardware/software technical restrictions support the description of this dependence. Observe that integration restrictions are always possible at the technical interface level.

- Product and Production Philosophy

The spectrum of products and part-products, the description of the design, drawing and of the production process, their connections and a technical description of the tools used in the whole chain form a basis for the comprehension of this philosophy. Unnecessary to say that we do not claim to give a complete solution to this description, our aim is to provide the problem with some simple (e.g. tree) structures and analyse the evolution of the problem description and its successes and/or failures.

- Flexibility of the organization

Suppose we have an approximated description of the firm structure, also on what concerns its technical power; it seems then not absurd to estimate the flexibility of the organization for a new formalized objective.
- CAD-system allocation

The machines can all or part of them be in one building, different cities, etc. This gives measures for cables and interfaces.

- Clearness of the task to be supported by CAD-systems

Through the description of the firm structure and application fields, if the tasks or areas are not yet determined, they can be deduced from the characteristics of the systems on the market by giving the highest priority to the better covered area (avoiding risks). Of course no one is here speaking of research areas but areas of competitiveness.

These parameters and their following levels do not yet claim for completeness. They are a first approach, still to be refined. A more detailed discussion of these parameters and the way of quantifying them can be found in /1,6/.

4. Technical Parameters

These parameters evolve from a technical and functional description to a computer configuration.

Aiming the first purpose, which is to choose the adequate CAD-system configuration, a reference configuration is proposed, through which a computer configuration can be automatically built. The user builds such a configuration using an interactive graphics module. This configuration is a first approach to the configuration needed from a functional point of view.

A detailed list of hardware and software components constitute the characteristics of the configuration through which performance measures and productivity factors can be calculated, also based on the organizational parameters.

Among them, there is detailed information (parameters) about:

<table>
<thead>
<tr>
<th>Host Hardware</th>
<th>Host Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-architecture</td>
<td>Operational System</td>
</tr>
<tr>
<td>Word size</td>
<td>Compiler</td>
</tr>
<tr>
<td>Communication processors</td>
<td>Database</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Application Programs</td>
</tr>
<tr>
<td>Bus capacity</td>
<td>Network, etc</td>
</tr>
<tr>
<td>Memory, etc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local Hardware</th>
<th>Local Software</th>
<th>CAD-Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor type</td>
<td>Emulators</td>
<td>Geometric Elements</td>
</tr>
<tr>
<td>Coordinate space</td>
<td>Interfaces, etc</td>
<td>Geometric Model</td>
</tr>
<tr>
<td>Controller</td>
<td></td>
<td>Object Manipulation</td>
</tr>
<tr>
<td>Color</td>
<td></td>
<td>Generation's Principle</td>
</tr>
<tr>
<td>Graphic display functions</td>
<td></td>
<td>Transformations, etc</td>
</tr>
<tr>
<td>I/O devices, etc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appendix 1a and 1b show the implementation of these technical interfaces.
5. Performance and cost parameters

When the technical parameters of a configuration are already set, if there exist components on the market which could build up an equivalent configuration, the start costs and the costs per year for this configuration could be estimated, based on real value calculation and ideal approximations.

The start costs are:

- Hardware and software investment
- Room preparation
- Cables and connections
- Firm actual state and CAD-introduction planning
- CAD-system's choice
- Introduction preparation
- Training
- Installation and integration
- Dead period
- Data input

The costs per year are:

- Personnel
- Training
- Data integrity and backup copy
- Material and power consume
- Hardware and software maintenance
- Insurance
- Capital interest
- Rent
- Depreciation

This representation of costs is available to the user as a menu interface in different levels of details according to the specific cost type.

Performance parameters are associated with methods which use them to obtain statements about the performance of a certain configuration under a certain environment. For this reason the performance parameters which are considered at the moment apply directly to some implemented methods concerning the relationship performance/cost.

These parameters are:

- Acceleration factors on drawing (Appendix 2)
  - description of a typical drawing
  - description of the technical parameters to solve the task

- Acceleration factors on activities (Analog to appendix 2)
  - description of the various phases of a process
  - description of the technical tools applied

- Benefit value
  - A benefit value is drawn for each alternative system based on a classification of the components of a CAD-system /1/ and on weights given to the components according to its importance for the application.
As one can see these parameters are based on experimental values.

6. Analysis Methods

There is still not a final formal definition for the analysis methods. These are a very broad and abstract field which will be progressively occupied by recognized support methods, which will in turn tend to determine their formal specification.

Without much concern on preciseness, analysis methods are then used to classify the elements of a system attributing to them its functional value (or behaviour) in the environment. In connection to our approach, analysis methods support research on organizational, technical and economic parameters by constructing empirical rules. If there exists a support theory for a problem, there is at first no need for analysis methods, except for those which empirically not theoretically find the adequate theory to the specific problem.

As a matter of fact the word analysis contains the meaning of resolution of problems by reducing them to tractable units. This is exactly what has been tried with regard to our parameter sets. Nevertheless, this has been done per trial and error approach, and do not still constitute a methodology for treating problems analytically.

An implicit subset of the analysis methods are the performance/cost methods, which will be now discussed:

- Smallest time method (Appendix 2)

A typical engineering drawing is divided into its elements (material specification, rotational parts, cuts, dimensioning, etc.) to which acceleration factors dependent on the available configuration are designated. The total acceleration factor is then given by the formula:

\[
TAF = \frac{\sum_{i=1}^{n} AF_i \times NE_i}{\sum_{i=1}^{n} NE_i}
\]

which calculates the mean acceleration factor for a drawing supported by a CAD-configuration. The acceleration factors \( AF_i \) for individual elements is based on measured values according to research studies /1/. \( NE_i \) is the number of elements 'i' of a drawing with corresponding acceleration factor \( AF_i \).

The difficulties encountered here are the composition of the set of elements, the corresponding hardware/software support to one element, the acceleration factor interval of an element and the value designated based on the hardware/software support. Experimental values are used based on common sense.

This method can be extended to treat acceleration factors for processes in general, by dividing the process in known tasks supported by hardware and software tools.

In general, if the time required for the manual production of a drawing is given and this task is accelerated through hardware/software tools by a given acceleration factor, the cost reduction can be easily calculated. Such simulation is also implemented.
- Benefit analysis (Nutzwertanalyse)

This method consists of:

Construction of a tree-structure representing the object (CAD-system configuration) to be analysed, where the leaves are the partial objectives (components of the system) to have their values compared.

The objectives are weighted according to their importance for the application, so that each level of objectives sum up to 100%.

The possible values of a partial objective are classified in a table or as a function, giving a note (N) (usually between 0-5 or 1-6) to the value.

To each alternative configuration a benefit value is calculated through the formula:

\[ BV = \sum_{i=1}^{n} N_i \times W_i \]

which calculates the sum of the partial benefits by multiplying the corresponding note \( N_i \) attributed to the value of the objective \( i \) times the accumulated weight \( W_i \) given by the product of the weights in the path.

The number of alternative systems entering the method should be reduced. This is done by stating some characteristics which the alternative systems must contain.

These methods of analyse aid the user/expert in the construction of speculations. The process of taking a decision, which has no means to be taken by the system, is also supported by allowing an interactive dynamic change and orientation based on actual parameters. For example, to search for gains in productivity, economic factors of cost and performance are considered interactively.

7. Relation to general design theory

According to Yoshikawa's general design theory /2/, we may define the following formulation example:

- The entity set \( S' \) contains the subsets:
  - \( S_1 \): organizational parameters (OP),
  - \( S_2 \): technical parameters (TP),
  - \( S_3 \): performance and cost parameters (PCP),
  each of them contributing with \( S_x \) entities of \( S' \).

- The abstract concept set \( T \) contains:

\[ T := \{ T_1, \ldots, T_n \} := \text{classification of concepts of entities,} \]
\[ \text{relationships between attributes,} \]
\[ \text{functions or structures of the entities.} \]

The abstract concept space is \((S_1, S_2, S_3, T_1, \ldots, T_n)\).

For the understanding of objectives and the disposition of parameters, one could think of Fig.1 as being the "region of possible solutions" \( S_e \), which concatenate \( S_1, S_2 \) and \( S_3 \) each respectively with dimensions \( M, N \) and \( K \) according to \( T_1, \ldots, T_n \). This is a set of entities which satisfies some requirements.
Fig. 1: Region of possible solutions (CAD-system configurations)

The concepts of function and attribute of entities are better visualised through values. Assuming the entity set $S'$, we have:

Attribute set $T^s$
- for $S_1$

$$T_s^f := \{ \begin{align*} T_{s1}^f & := \text{electronic application;} \\ T_{s2}^f & := \text{microfilm;} \\ T_{s3}^f & := \text{between 10 and 30 designers; etc} \end{align*}$$

for $S_2$

$$T_s^g := \{ \begin{align*} T_{s1}^g & := \text{2D line model;} \\ T_{s2}^g & := \text{3D volume model;} \\ T_{s3}^g & := \text{1 Host, 4 Wkst, 1 Plotter;} \\ T_{s4}^g & := \text{etc} \end{align*}$$

for $S_3$

$$T_s^g := \{ \begin{align*} T_{s1}^g & := \text{interaction is better than batch;} \\ T_{s2}^g & := \text{break even point;} \\ T_{s3}^g & := \text{benefits; etc} \end{align*}$$

Function set $T^f$

$$T^f := \{ \begin{align*} T_1^f & := \text{increase the production ratio to xx%;} \\ T_2^f & := \text{reduce the redundancy of information;} \\ T_3^f & := \text{improve the quality of drawings;} \\ T_4^f & := \text{accelerate a project's span time; etc} \end{align*}$$

Let us assume a specification $T = T_s^f \cap T_s^g$. 

210
If \((T_{x1} \cup T_{x2})\) satisfies \(T_{y1}\) and \((T_{z1} \cap T_{z2})\) satisfies \(T_{z1}\), we have the region of solution:

\[
S_x := \forall S_x \ (S_x \in \{(T_{x1} \cup T_{x2}) \cap T_{y1}\}) \rightarrow (T_{z1}(S_x) \rightarrow T_{z1})
\]

It should be pointed out that the meanings between attribute-attribute, attribute - function and function - function must be clear and at least supported by estimation approaches.

One challenge is to build a self-explanatory interface, where user and expert manipulate their data and achieve their grades of satisfaction.

8. Implementation approaches

The concepts presented are being implemented based on two different approaches. Both approaches are realized under Unix on a PCS Cadmus 9230. The first approach to the implementation of the system described consists of an interface programmed to allow graphic interactivity through menu handling built on top of GKS/10/ and to manipulate data contained in a relational database (CORAS)/5/.

Fig.2: Structure of the first implementation approach
The user may proceed as follows:

- enter values to the
  1. organization parameters
  2. actual and the planned configuration
  3. hardware and software components
  4. economic methods

( Some plausibility tests are set for the configuration )

- search for actual CAD-Systems

- ask for the calculation of configuration's costs and benefits

With the set of classified alternatives the user may take his choice.

Although there are many gates of interaction along the manipulation of the system, four basic interfaces are to be distinguished as shown in Fig.2. They build in a certain sense the input and the partial outputs for continuous manipulation of data and methods.

Along the development of the system many structural problems have occurred, which report to knowledge misunderstanding and mixtures and also to the use of immature (not commercial) hardware/software tools. The overcoming of these situations are not always very efficient, as it was the example of having to use direct access Fortran files to compensate restrictions of the database. Nevertheless, the possible amount of data and methods to be handled at present seems to be useful for prototype testings.

On what concerns the classification of hardware and software of a CAD-Configuration, around 300-400 technical parameters can be used for comparison. A technical parameter can be seen as the path from the root (CAD-configuration) to the leaf, which contains the value for the path. Tests with real values are being carried out, although the integration of performance methods do not consider the whole set of parameters.

One still missing point in this implementation is the availability of some statistical methods to arrange classes of information for use by closely repetitive processes. It seems nevertheless that nothing could hinder such an extension.

The second approach, which is in effect being implemented parallel to the first one, uses the high level language Prolog /4/. This implementation does not yet consider the whole CAD-Model as defined.

The main concern in this approach is the representation of the two sets of parameters and relationships between them. As it has been shown above, there is no pure mathematical (analitical) formalization for this problem. However such relation exists and some people use it to evaluate and choose the best CAD-System for each particular case. It is clear that this relation is some kind of knowledge. Therefore the expert system (knowledge based) approach is most suitable for our case.
Fig. 3: General structure of the decision support system

The problem of choosing a proper CAD-system, according to the user requirements, comes to finding out the relationship between two sets of parameters: organization parameters (OP) and technical parameters (TP). Thus we can consider a decision support system (expert system), which may have:

- database of all possible CAD-system configurations (or classes of CAD-systems) with their technical parameters;
- set of OP for CAD-systems in general;
- relation OP \( \leftarrow \) TP, i.e. a CAD-system configuration (class of configurations) corresponds to each set of OP with assigned values.

With respect to the above mentioned purposes, the system contains the following blocks:

1) knowledge representation block (knowledge about two types of parameters and relationship between them);
2) user interface - for interacting with the user
3) expert interface - by means of this block the knowledge about CAD-systems is extracted from experienced people in this field (CAD experts) and is stored in the knowledge representation block.

The necessity of many parameters and random access to them dictates the idea of using a relational database for storing technical parameters of a number of CAD System configurations and possible combination of OP with given values.
A well known method for representation of relationship between parameters, which can not be represented analytically (with formulas) is production system or simply said, set of rules, which in our case, have the following form:

\[
\begin{align*}
&\text{OP}_1(V_1) \text{ And } \text{TP}_1(P_{11}, \ldots, P_{1m}) \text{ Or } \\
&\text{OP}_2(V_2) \text{ And } \text{TP}_2(P_{21}, \ldots, P_{2m}) \text{ Or } \\
&\ldots \\
&\text{OP}_k(V_k) \text{ And } \text{TP}_k(P_{k1}, \ldots, P_{km}) \text{ Or }
\end{align*}
\]

here:

\(\text{OP}_1, \ldots, \text{OP}_k\) are the organization parameters. \(V_1, \ldots, V_k\) are values of organisation parameters including "don't care" or limits.

\(P_{ij}\ i=1, \ldots, n\ j=1, \ldots, m\) are CAD technical parameters. "n" is the number of possible configurations corresponding to the one and the same OP set. "m" is the number of technical parameters.

In the general structure of the decision support system (Fig.3), an additional block (menu) is added to make the system domain independent. This block is a hierarchical structure of menus (for OP and TP), which is used for interaction with the user and also defines the internal structure of the parameters (OP and TP).

The domain independent part of the system (blocks 1,2,3,4) manipulates the menus, asking the user or expert and constructs the structure of two databases (blocks 2,3) with values of the asked parameters obtained from the answers.

The block 4 (expert interface) is used only at the development of the system to fill the databases. Filling the databases is performed by defining rules, which enable the system to make inferences and also define the structure of parameters (OP and TP).

The same menu structure (block 5) is used for the dialogue with the user. In this case the system constructs a set of OP's and tries to match them with IF-condition part of some existing rule. In case of matching, the right part of the matched rules (configurations with parameters) is printed as a solution.

If there are more than one solution, the benefit analysis method can be used to find the best one.

The user can also specify a CAD-system configuration with some parameters and obtain from the system all CAD-system configurations, in this class with all parameters (right part of the rule).

As a tool for this implementation we consider the language Prolog. It provides all needed mechanisms (relational database and inference) and makes the implementation easier.

It is important to note that accessing the rule in the Prolog database is performed automatically by built-in mechanisms of pattern matching and finding alternative solutions by backtracking.

This implementation supports at the moment interfaces (without graphics) menu structured and database mechanisms. In this stage the system is ready to be filled with real knowledge supplied by experts.

For better interaction, graphics can be used in the interface blocks. Using graphics with Prolog also gives some advantages especially for
implementation of highly interactive interfaces and shows already some interesting insights for future projects. This work is being done based on a C-interface for the GKS - Prolog communication, and it will be part of this second implementation approach, improving still more the interactive character of the description and the analysis.

9. Conclusions

In the present paper the general structure of a CAD-model has been proposed and some formalisms have been introduced as a basis for the design of implementations of such a decision support system for CAD evaluation and choice purposes.

Two different approaches for its implementation have been described; one of them based on Prolog.

The first implementation approach supplies the user dialog with a highly interactive graphic interface based on GKS for describing the CAD-system configuration parameters (appendix 1a and 1b) and with menu oriented interfaces for the description of user's environmental requirements such as firm organizational description and types of applications, and also for the description and/or acceptance and use of performance and cost parameters and methods for analysis.

Structural concepts for implementation and domain dependent knowledge on CAD-evaluation have been analysed, based on the first implementation and on the tools available for the second implementation.

Since the second implementation is based on tools for the construction of a system approaching the handling of knowledge structures, its chance of turning out to be a more efficient and future oriented system increases. The additional use of graphics-GKS with Prolog also improves the evaluation system and opens new possibilities for research.

It is also expected that based on its usage the common knowledge about CAD-systems will turn out clearer.

10. References

/1/ Encarnacao, J.L. et.al.
GI-CAD/Handbuch: Auswahl und Einführung von CAD-Systemen
Springer Verlag 1984, ISBN 3-540-13797-1

/2/ Yoshikawa, H.
General Design Theory and a CAD-System
in: Man-Machine communication in CAD/CAM
North Holland, IFIP 1981

/3/ Spur, G. and Krause, F.-L.
CAD-Technik
Hanser Verlag 1984, ISBN 3-446-13897-8
4/ Clocksin, W.F. and Mellish, C.S.  
Programming in Prolog  

5/ Encarnacao, J.L. and Schlechtendahl, E.G.  
Computer Aided Design  

6/ Encarnacao, J.L.  
Interfaces of a CAD-System with its application environment  
in: CAD Systems Framework (K. Bo, F. Lillehagen Editors)  
North Holland, 1982

7/ Encarnacao, J.L. and Messina, L.A.  
A system simulation technique for the technical evaluation and economic  
jjustification of CAD-Systems  
VDE Verlag, Proceedings of CAMP'83 Berlin

8/ Eigner, M. and Maier, H.  
Einstieg in CAD - Lehrbuch für CAD-Anwender  
Hanser Verlag 1985, ISBN 3-446-14118-9

9/ Kasprzak, W.  
Bemerkung zu der ersten Version des Papers  
Private communication - Warszawa, 26.05.85

10/ Enderle, G., Kansy, K. and Pfaff, G.  
Springer Verlag, ISBN 3-540-11525-0, 1984
Appendix 1a:

Detail from the technical parameters interface
(Hardcopy)
Appendix 1b:

Detail from the technical parameters interface

(Hardcopy)
## Appendix 2:

**Interface for description of a typical drawing**

*(Hardcopy)*

<table>
<thead>
<tr>
<th>NE</th>
<th>AF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification of a drawing</td>
<td>(2.95)</td>
</tr>
<tr>
<td>Test, quality and delivery regulations</td>
<td>( )</td>
</tr>
<tr>
<td>Manufacturing regulations</td>
<td>( )</td>
</tr>
<tr>
<td>Material specification</td>
<td>( )</td>
</tr>
<tr>
<td>Raw material specification</td>
<td>( )</td>
</tr>
<tr>
<td>Surface texture symbols</td>
<td>( )</td>
</tr>
<tr>
<td>Additional specifications</td>
<td>( )</td>
</tr>
<tr>
<td>Contour elements</td>
<td>(1.80)</td>
</tr>
<tr>
<td>Dimensionings</td>
<td>70 (6.20)</td>
</tr>
<tr>
<td>Tolerance dimensioning</td>
<td>4 (5.00)</td>
</tr>
<tr>
<td>Geometrical and positional tolerance</td>
<td>(5.60)</td>
</tr>
<tr>
<td>Hatch-areas</td>
<td>3 (3.40)</td>
</tr>
<tr>
<td>Scale specifications</td>
<td>( )</td>
</tr>
<tr>
<td>Dimensioning arrows</td>
<td>123 (3.50)</td>
</tr>
<tr>
<td>Auxiliary lines</td>
<td>12 (4.00)</td>
</tr>
<tr>
<td>Lines</td>
<td>80 (1.30)</td>
</tr>
<tr>
<td>Special symbols</td>
<td>( )</td>
</tr>
<tr>
<td>Macros</td>
<td>(2.57)</td>
</tr>
<tr>
<td>Texts</td>
<td>( )</td>
</tr>
<tr>
<td>Unwinding and/or revoluted surfaces</td>
<td>(7.50)</td>
</tr>
<tr>
<td>Views</td>
<td>( )</td>
</tr>
<tr>
<td>Straight sections</td>
<td>(1.75)</td>
</tr>
<tr>
<td>Penetrations</td>
<td>( )</td>
</tr>
<tr>
<td>Separate details</td>
<td>(4.50)</td>
</tr>
<tr>
<td>Enlarged details</td>
<td>2 (3.00)</td>
</tr>
<tr>
<td>Symetric elements</td>
<td>(5.97)</td>
</tr>
<tr>
<td>Rotated and/or translated elements</td>
<td>6 (6.00)</td>
</tr>
<tr>
<td>Copied elements</td>
<td>( )</td>
</tr>
<tr>
<td>Standard grade (1-3)</td>
<td>( )</td>
</tr>
<tr>
<td>Variants</td>
<td>(5.10)</td>
</tr>
<tr>
<td>Datatransfer to: - Methods of calculation</td>
<td>1 (4.00)</td>
</tr>
<tr>
<td>Production resources</td>
<td>( )</td>
</tr>
<tr>
<td>.Parts list preparation</td>
<td>( )</td>
</tr>
<tr>
<td>.Production plan preparation</td>
<td>( )</td>
</tr>
<tr>
<td>-NC-Maschine</td>
<td>( )</td>
</tr>
<tr>
<td>-different types of drawing</td>
<td>( )</td>
</tr>
<tr>
<td></td>
<td>3.56</td>
</tr>
</tbody>
</table>
A PICTORIAL OVERVIEW OF SOLIDS MODELING AND RENDERING

Andries van Dam

Computer Science Department, Brown University, Providence, Rhode Island, USA

1. INTRODUCTION

In the proceedings of the 1982 CERN School of Computing, I concluded my article "An Overview of Interactive Computer Graphics and its Application to Computer-Aided Design" with a prediction of the major impact that networks of graphics-based workstations would have on the practice of CAE/CAD/CAM, particularly in terms of the user interface. There has, in fact, been an explosion of interest in and use of graphics-based personal computers and workstations, especially in the last few years, and it is fair to say that workstation with compute power typically greater than that of a VAX/780 are now the norm. Older time-sharing systems with graphics terminals still survive today because software has not yet been ported to more modern workstations, and because central databases are more easily implemented on such machines than on a file server accessed over a network.

In the article I described how both electronic and mechanical CAD were database-centered and consisted of interactive (design) front-end programs which created the database and back-end programs that processed them. For electronic design (E-CAD) such "post-processors" include simulation, verification, placement and routing. For mechanical design (M-CAD) it includes analysis and simulation, process planning, and the creation of manufacturing data and of N/C tapes.

Another theme underlying both E-CAD and M-CAD is that catching errors through interactive simulation, verification and analysis at the earliest possible opportunity saves costly reworking at later stages in the design/manufacturing pipeline. (A similar phenomenon has also been documented for decades in the software engineering literature.) Such interaction requires both adequate compute and memory power and a responsive user interface.

Design of printed circuit boards and VLSI chips is done well enough on today's generation of workstations with their medium- or high-resolution color displays with 4 or 8 planes per pixel and 1-2 Mips of compute power. Indeed, much electronic design is even done on personal computers, whose compute power and graphics add-on boards turn them into near-workstations. Only for such batch compute processes as simulation, placement and routing is a workstation compute power inadequate; here a link is required between the personal computer or workstation and a mainframe or special-purpose engine which handles such compute-intensive jobs. While E-CAD is relatively stable, M-CAD is undergoing a major transition. M-CAD previously concentrated on computer-assisted drafting, two-dimensional constructions and wireframe (or edge-list) representations of solid objects, in the form of polygonal approximations. The orientation is now shifting towards interactive solids modeling. Solids modeling is most often done by means of constructive solid geometry, which defines the boolean union, intersection and difference operators over a set of solid primitives that are the extensions of two-space polygons, rectangles, cones and polynomial curves. These solids include spheres, cuboids, cones, pyramids, objects obtained by rotation or translation of a profile or a closed curve respectively, etc., as well as solids described with closed surfaces modeled with Bezier splines or non-uniform rational B-splines. Solids modeling has gone from laboratory experimentation to production use not only in the aerospace and automobile industries but also in high-energy physics. At CERN, for example, the MATRA DATAVISION EUCLID system has been used effectively for various pieces of mechanical design of the LEP, both large-scale and small-scale. Solids modelers still have significant difficulties in dealing with free-form surfaces, fillets and other aspects of non-mathematical, real-world objects, but they have earned their place in the arsenal of designer's tools.

Current solids modelers are also hampered by running, by and large, on timeshared VAXes or IBM mainframes, which provide poor response for the extremely compute-intensive solid Boolean operations, and pre-viewing only in wireframe mode. Today's workstations of the SUN/Apollo/DEC/HP class also do not provide enough compute power or enough graphics power for genuinely interactive solids modeling. In other words, whether on timeshared super-minis and mainframes or on workstations, solids modeling is primarily a batch-style design process.

During 1987 there will be a quantum improvement in the amount of compute and graphics power available to an individual at a workstation. A new class of
"graphical supercomputer" workstation will become available, with computer power roughly in the 15-25 mflop, 20-40 mflop range and the ability to display one to several hundred thousand polygons per second. Each such polygon may have several hundred pixels and will be rendered with full hidden surface elimination. In addition, lighting and shading models will determine the color of each individual pixel as a function of light sources in the scene and material properties of the object which the polygon approximates. These workstations are designed to address problems arising in high-end science and engineering in such areas as interactive data analysis, interactive solids modeling, molecular modeling, computational fluid dynamics, etc. Today such large-scale problems can be tackled only by using a supercomputer to do calculations and then to generate a (structured) display file which is transmitted to a high-performance graphical workstation such as the Silicon Graphics IRIS or Apollo DSN50. The new workstations will allow this two-step process to be combined within a single, ultra-high-bandwidth system, to provide both more cost-effective and more responsive computing, coupled with much greater realism in the rendition of images. It will be possible with such systems to compute the kinematics of relatively complex objects and display a fairly realistic rendering of it in real time. Fig. 1 shows an example of object and rendering complexity that will be feasible with this new generation.

In the next section I will give a brief pictorial overview of solids modeling and shaded image generation that will be made possible by the new generation of workstations.

2. A PICTORIAL OVERVIEW

Fig. 2 shows a rather fanciful rendering of a simplified milling machine. How did we go about creating this image? Fig. 3 shows a "parts explosion" of its solid primitives: cylinders, cuboids, (truncated) pyramids, etc. Fig. 4 shows the set of primitives that a simple solids modeler might support; many of these were, in fact, used to model the milling machine. Figs. 5-9 show how a piece of the milling machine, the head consisting of a cylindrical housing and a bit, was modeled using Boolean operators. These are wireframe rather than shaded renderings because the interactive solids modeler allows such construction in real time only on a dynamic vector display; shaded images are rendered on a VAX/780 driving a simple frame buffer. Fig. 5 starts the construction process with a sphere that has been distorted to form an ellipsoidal solid. Fig. 6 uses Difference with large cuboids to remove the top and bottom pieces of this volume to create an ellipsoidal cone. Fig. 7 shows another Difference, this one removing edges by cylindrical portion from the ellipsoidal cone. Note that, as with all Boolean operators, the placement of the two solid operands relative to each other is crucial; interactive controls for doing the placement precisely (and mathematically correctly) are part of what is needed to make interactive solids modeling useful. Fig. 8 shows how the one-way grooved cone is Difference with the same cylinder at a new position and angle to add the second groove; two more grooves are added and Fig. 9 shows how the finished cone is Unioned to the attached cylindrical stem. Fig. 10 shows the wireframe results of modeling all pieces of the milling machine using the Boolean operators.

Having created the object in the three-dimensional world coordinate system, we now make a sequence of successively more realistic renderings of it by means of a high-level graphics package that incorporates lighting and shading models. First we remove visual clutter by doing "hidden edge removal" (also called "visible surface determination"), as shown in Fig. 11. We zoom in on the head in Fig. 12 to show that for wireframe display (whether with or without hidden edge removal) we must make a polygonal approximation to the mathematically accurate internal representation in terms of Boolean operators and solid primitives. The more polygons we use, the better the approximation but the more compute and display power are required. Fig. 13 shows a "flat shading" rule applied to all polygonal faces; Lambert's cosine law for diffuse reflection is used to shade proportional to the cosine of the angle between the surface normal and the direction to each point light source. The resulting picture is flat and chalky and not very realistic, as shown in the detail of Fig. 14. In particular, edges between adjacent polygons are shown very clearly and this makes the polygonal approximation all too obvious, even when polygons are very thin and closely spaced together, as in the grooves of the bit. The solution is to spend more compute time to interpolate colors across neighboring polygons using "Gouraud shading", as shown in Fig. 15. This figure also shows the addition of specular highlights, obtained via a rule that takes the cosine between the angle of reflectance and the view angle to a highlight using "Phong shading". Phong shading yields a rendering that is realistic enough for
most purposes, though additional effects such as transparency and shadows could be used to good advantage. Such additional effects are typically provided by "ray-tracing", a process that may consume days of computation on a VAX/780-class machine. Fig. 15 also shows the effects of anti-aliasing (minimizing the effects of discrete sampling so evident in the wireframe rendering of Fig. 11). For most interactive solids modeling purposes, flat shading gives adequate realism for moving images, and the more computationally expensive processes of Gouraud or Phong shading can be reserved for publication-quality images.

3. SUMMARY

Realistic and pseudo-realistic images will quickly displace wireframe representation for real-time modeling in many situations in science and engineering, now that the amount of computing power can be dramatically increased due to the migration of supercomputing power to personal workstations. This will be achieved with a combination of improvements in CPU and display processor architecture, thanks to (semi-) custom VLSI designs. New applications, especially in 3D M-CAD and high-end science and engineering, will benefit greatly from this new interactive compute-and-visualize capability.
FROM SHALLOW TO DEEP KNOWLEDGE

Henk de Swaan Arons

Dept. of Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands

Abstract
The Delfi project started at Delft University of Technology in 1982. Within this project the expert system shell Delfi-2 [1] has been designed and developed, and which has been modelled on the Emycin system. Delfi-2 has been successfully used in applications of various kind. Yet, its knowledge representation heavily leans on a rule base that only allows the use of shallow knowledge. Many applications, however, require deep knowledge instead. For this reason, a new knowledge engineering environment Delfi-3 [2], has been designed and partially developed. It is based on a knowledge representation model Damor [3] and is a semantic net of frame-like objects that allows a variety of internal inferences.

1 Introduction
Delfi-2 is a tool that can be used for building expert systems. For this reason it is called an expert system shell or an expert system building tool. Its design is based on Emycin [4].

Expert systems that have been developed with Emycin-like systems, and thus with Delfi-2, consist of two separated components: the knowledge base and the consultation system. The first contains the domain knowledge. This is both factual and heuristic knowledge. The consultation system allows the user to get expert advice from the expert system.

Although the knowledge base of Delfi-2 in fact consists of two parts (a rule base and a declarative part) the actual knowledge is contained in the rule base. This approach has both strong and weak points. First, a production rule allows the expert to express specialized knowledge in the form of a relation between a premise and a conclusion, augmented with a degree of certainty. This appears a powerful mean to represent this kind of knowledge.

On the other hand, a production rule is not suited to describe what is called deep knowledge: the objects that are relevant to the domain, the sometimes trivial relations (the semantics) between objects, and the hierarchical structure that is almost always present. This kind of knowledge requires another kind of representation. Delfi-3 will provide the means for this.
Next section presents a brief description of the Delfi-2 system, its knowledge base and its inference engine. The same is done in section 3 with respect to the Delfi-3 system. Some emphasis will be put on Damor, which provides the mean to represent deep knowledge. In section 4 some near-future tasks are mentioned.

2 A brief description of Delfi-2
Delfi-2 consists of two parts, a knowledge base and a consultation system. In figure 1 its components are shown in more detail.

![Diagram of Delfi-2 components]

Figure 1. The main components of Delfi-2

2.1 The knowledge base
Next the two parts of the knowledge base are considered in more detail.

2.1.1 The declarative knowledge
The knowledge base is stored in separate files with extensions .par and .rul respectively. The declarative knowledge shows some similarity with the AI-concept frame. It allows the expert to split up the knowledge domain into one or more objects which are called contexts. Each context has a set of characteristics, called parameters. An example of a context with some parameters is given in figure 2.
Context: tenant
Parameters: marital_state (* TXT, INITIALDATA, ASKFIRST *)
           income (* INTEGER, INITIALDATA, ASKFIRST *)
           has_children (* BOOLEAN *)
           granted (* BOOLEAN, GOAL *)

Figure 2. An example of a context with some parameters in Delfi-2

Each parameter has some additional information concerning its type (TEXT, INTEGER, REAL, BOOLEAN), its LEGALVALUES (e.g. a tenant could be either single, married or divorced), how the inference engine has to operate to get one or more values assigned to a parameter (e.g. in order to get the integer value of the income of the tenant, the user has to be asked for it in the initial phase of the session). Thus, in the .par file not only all contexts and their parameters are defined, it also contains control information for the inference engine.

Production rules may only use contexts and parameters which are defined in the .par file.

2.1.2 The rule base
A method widely used to represent knowledge are production rules. In Delfi-2 the emphasis of the knowledge representation has been put on production rules. These rules have an IF-THEN structure, i.e. a number of conditions is related to a number of conclusions which could in some instantiated situation be concluded from the conditions. An example of a Delfi-2 rule is given in figure 3. The first part (a) shows how the rule is represented internally. In (b) the same rule is given, now in translated form.

A production rule has the following elements:

- predicates and actions (e.g. same, notsame, between, oneof; conclude, execute);
- one or more contexts;
- parameters, expressions of parameters or function calls;
- lists of constants;
- one or more certainty factors (depending on the number of conclusions).
Two points are worthwhile to mention here, which have proved to be particularly useful in technical applications: in a rule there is the possibility to use either numerical

(a) IF
same oscillator amplitude constant;
notsame oscillator driving_force yes;
THEN
conclude oscillator model \( d^2x/dt^2 + kx = 0 \);
1.00

(b) IF
1.0 the amplitude of the oscillator is constant
2.0 the oscillator has no driving force
THEN
it is definite (1.0) that the oscillator model
is \( d^2x/dt^2 + kx = 0 \)

Figure 3. An example of a Delfi-2 rule. (a) is the internally used form, (b) the translated form to be used in interaction with the user

expressions or function calls. In a rule, quite often the result of a numerical expression has to be dealt with. An example is given in figure 4. In many applications, however, there is a lot of application software already available (written in some programming

IF
same tenant marital_status married;
OR
same tenant cohabitation yes;
between tenant rent 217 700;
lessthan tenant inc 40758;
same tenant working_partner yes;
lessthan tenant [inc+((inc_partner-2000)/2)] 40758;
notsame tenant exception yes;
THEN
conclude tenant subsidy_granted yes;
1.00

Figure 4. A numerical expression used in a production rule

language), which has to be used by the expert system in order to get a good performance. Then the expert system must be able to make use of this application software, see also figure 5.

In figure 4a kind of fictive income is calculated, the result of which is only used in the concerning rule and is not stored. If the calculation is more complex (the calculation in figure 4a implies some peculiarities which can only be overcome by introducing some control structures; then a pocket calculator function does not satisfy anymore), then a
IF
  same tenant marital_status married;
OR
  same tenant cohabitation yes;
  between tenant rent 217 700;
  less than tenant inc 40758;
  same tenant working_partner yes;
  less than tenant tot_inc(inc,inc_partner) 40758;
  not same tenant exception yes;
THEN
  conclude tenant subsidy Granted yes;
  1.00
  execute tenant tabel_A(tot_inc(inc,inc_partner),rent);

Figure 5. A function call and a procedure call in a production rule

function can be called. The same happens in the conclude part of the rule, where the amount of subsidy is found by using the appropriate table.

2.2 The consultation system
The consultation system consists of three parts, of which the inference engine is the most important one. It is a backward chaining mechanism (more recent releases of Delfi-2 also support forward chaining). Its main task is to determine the value(s) of one or more goal parameters. In order to find them it needs to find many other parameter values. For example, in order to be able to determine whether a tenant could get a rent subsidy (the ultimate goal of a consultation) it needs to know his income, marital status etc. The inference engine has four possibilities to find these values:

- a value could be deduced from the production rules
- a value could be asked from the user;
- values could be retrieved from external data bases;
- by calling of functions.

In most cases the inference engine will attempt to deduce the values from the rule base. However, as mentioned before, information in the _par_ file can influence this mechanism.
Both inference mechanisms are well-known and will not be discussed any more here.
For many applications the explanation facilities are quite useful; they provide the user with an insight in questions such as why a question is asked and how certain facts have been deduced. These are the so-called WHY and HOW facilities.
Further, Delfi-2 offers to the user the possibility to track the complete deduction process on various levels. This trace facility is mostly used by the knowledge engineer.

2.3 Applications with Delfi-2

Delfi-2 has been used and is still in use in a great variety of applications. Figure 6 gives an overview of the companies in which Delfi-2 is in use, the type of applications and some remarks.

<table>
<thead>
<tr>
<th>Company</th>
<th>Project</th>
<th>Nature</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLM</td>
<td>B747/ATA-21</td>
<td>Fault-finding in a technical environment with Delfi-2</td>
<td>One of the first big knowledge bases</td>
</tr>
<tr>
<td>SSDZ</td>
<td>Thyroid/Anemia</td>
<td>Interpretation of clinical-chemical test data</td>
<td>No interactive use of Delfi-2: input data-&gt;report</td>
</tr>
<tr>
<td>CWI/RUL</td>
<td>Hepar</td>
<td>Diagnostics of liver diseases</td>
<td>Emphasis on validation</td>
</tr>
<tr>
<td>Delft Univ. Dept. of Materials</td>
<td></td>
<td>Selection of materials</td>
<td>Emphasis on the use of data bases</td>
</tr>
<tr>
<td>Delft Univ. Dept. of Electronics</td>
<td>Clusan-1</td>
<td>Analysis of image data</td>
<td>Emphasis on use of statistical software on other computers</td>
</tr>
<tr>
<td>National</td>
<td>TARIFFEUR</td>
<td>Fares int'l good transport</td>
<td>Use of data bases, purpose: operational system</td>
</tr>
<tr>
<td>Delft Univ. Dept. of Mechanical Engineering</td>
<td>Plexus</td>
<td>Diagnosis and therapy of neck-bone injuries</td>
<td>Purpose: operational system in revalidation centres</td>
</tr>
</tbody>
</table>

Figure 6. An overview of some applications with Delfi-2.

3 A brief description of Delfi-3

A weak point of Emcyin-like systems is their limited knowledge representation. Production rules provide an excellent mean to represent heuristic knowledge. However,
in any domain there is a lot more to know than what is present in the rule base. For instance, the rule given in figure 4 (it is not really a heuristic rule) explicitly states in which case a tenant could get a rent subsidy, but for a careful human reader there is more knowledge contained in it than this. We know that a tenant has a name, that a working partner is -mostly, so by default- a woman, that the income of the tenant is -also by default- higher than that of the partner, that the tenant and the partner have the same address, etc. For this kind of knowledge there is no place in Emycin-like systems since production rules can imply relations without being aware of background knowledge. Production rules, therefore, are said to represent shallow knowledge. The background knowledge, i.e. all relevant knowledge about the domain and its environment, is called deep knowledge. For this reason, a new expert system shell must have a powerful representation language as a kernel. For Delfi-3 this kernel is Damor [3], of which a preliminary version has already been developed.

In practice, production rules have demonstrated to be an adequate mean to represent heuristic knowledge. This is the reason that Delfi-3 will again offer the possibility to represent (mostly heuristic) knowledge by means of production rules. The production rules are based on the descriptive knowledge as defined in Damor. Since Damor is the most striking difference with Delfi-2, we shall consider its basic concepts.

3.1 Damor in a nutshell

Damor is a data model for knowledge representation. It is a combination of frame-based objects and a semantic net. The main components are objects and relations.

Objects are split up in Definition Objects (DOBJ) and Individual Objects (IOBJ). The first define the objects relevant to the domain to be modelled. Examples are automobile, machine, product etc. IOBJs are instantiations of DOBJs: My_Volvo and His_Lada are instantiations of the DOBJ automobile. A DOBJ is characterized by one or more attributes: price, country_of_origin, kind could be attributes of the DOBJ product.

Attributes can have several facets, of which the TYPE facet always needs to be present. For instance: price is of type INTEGER, and country_of_origin could be of type TEXT. Another is the LEGAL facet denoting a list of legal values: e.g. Switzerland, Holland, Japan, West-Germany. There could also be a DEFAULT facet, in this case Japan is a reasonable default value. Other important facets are VALUE (the value(s) of an attribute in an IOBJ), IF-NEEDED (defining the actions that have to be carried out to determine the value(s) of the corresponding attribute of an IOBJ) and IF-ADDED (defining the actions that have to be carried out in case the value(s) of the corresponding attribute of an IOBJ have been determined).

Since real-world problems tend to be quite complex, an application has quite a number of DOBJs involved and of course many more IOBJs. According to the actual situation in many applications, the DOBJs mostly are hierarchically structured by using the super
connection. The DOBJ man, for instance, could be connected to the DOBJ human implying that man inherits all information from human.

In this way an important part of the domain knowledge can be modelled: DOBJs represent relevant objects with their characteristics and can be built up in a hierarchical manner. However, a hierarchy is not the only relation between the various objects and therefore DAMOR offers the ability to represent all kinds of relations between DOBJs and also relations can be related by relations.

There are two kinds of relations: Definition Relations (DREL) and Individual Relations (IREL). The first define the relations relevant to the domain to be modelled. For instance, a DREL could describe when one automobile is considered to be a faster than another. This DREL thus relates two DOBJs, both of type automobile. The relation itself is defined in the body of the DREL, called the VIEW, in terms of attributes of both DOBJs. The DREL always_faster concerning DOBJ1 and DOBJ2 could be defined as: [(DOB1.velocity > DOB2.velocity) AND (ORD(DOB1.acceleration)) > (ORD(DOB2.acceleration))], in which it is assumed that velocity is of type INTEGER and acceleration is of type TEXT with the LEGAL list: (poor, moderate, good). The IREL always_faster1: always_faster(My_Volvo, His_Lada) is found true, since My_Volvo can go faster and accelerates better than His_Lada.

In this respect two remarks are worthwhile. First, an important point in Damor are the inferences that it should be able to make. Although some easy inferences are already possible (inheritance, for instance, and also some others), this item is of utmost importance for Damor itself and for any system that makes use of it, such as Delfi-3. Therefore, the inference engine within Damor is one of the key research items.

The second remark concerns the representation of heuristic knowledge: should it be done in Damor or in a rule base that uses the knowledge that is defined in Damor? Although it would be possible for Damor to represent heuristic knowledge (e.g. with the help of If-Needed facets), it is found that utilising a rule base would be more appropriate. For this reason, a few words will be spent to Delfi-3's rule base and how it could cooperate with Damor.

3.2 The rule base

As with objects and relations there are definition and individual rules. A definition rule defines a prototype event in terms of objects and relations. It consists of three parts: the declaration block, the condition block and the conclusion block. The declaration block specifies the definition objects the rule will be using in both the Condition Block and the Conclusion Block, referring to the objects that are defined in Damor. The Condition Block contains a boolean expression in terms of attributes of definition objects and definition relations between objects. An example of a Definition Rule is given in figure 7.

238
3.3 The inference engine

Since Delfi-3 is a combination of Damor (with its own inference capabilities) and a rule base (for which several inference techniques are already known and developed), there is the need of having a third inference engine that controls which part will get control to acquire certain facts. Some attributes could easily get their values within Damor, other facts could better be gained by inspecting the rule base. This approach implies that both Damor and the rule base will very frequently be used, calling each other all the time. So, Delfi-3 will be characterized by a hierarchical control structure: the inference engines in Damor and in the rule base, and -on top- what is called the master inference engine. For more details see [2].

4 Further prospects

Delfi-3 is not yet completed. A preliminary version of Damor has been developed, but it needs further refinements. Specifications of the design of a production rule have been started, and the first implementation including the inference engine will soon take place.

There are many other facilities that need to be developed such as: an adequate user interface, integration of Delfi-3 and (relational) databases, compilation of the knowledge base and advanced explanation facilities. In this respect previous experiences with Delfi-3's ancestor, Delfi-2, could make some design decisions much easier.

The experiences with Damor thus far stem quite hopeful with respect to the usefulness of the ultimate Delfi-3. Damor has already been used to describe a Flexible Manufacturing System (machines, orders, jobs, tools and pallets are relevant objects in this domain) including its semantics. This work has been carried out in the context of
the Esprit project 809 "Advanced Control Systems and Concepts in Small Batch Manufacturing"[5], in which Delft University of Technology is involved.

5 Acknowledgments
The author would like to acknowledge the many colleagues and students inside and outside Delft University of Technology who have been involved in the Delfi project and have contributed considerably to its present results. In this respect Peter Lucas of the Centre for Mathematics and Computer Science in Amsterdam certainly deserves to be mentioned for his excellent contributions to Delfi-2, and Willem Jonker for his creative work on Damor. This research has been made possible by the Dutch Ministry of Public Health that has granted the Medes project, which is being carried out in cooperation with Groningen Academic Hospital.

References
TRENDS IN ARCHITECTURES

L.O. Hertzberger

Computer Science Dept. (FVI), University of Amsterdam, The Netherlands

Abstract

As a result of the impact of modern VLSI technology we are faced with possibilities we are not able to fully exploit at the architecture level. A more scientific approach towards the design of instruction sets of CPU's and a better understanding of the role parallelism plays in architecture might offer a solution. Thorough research has to lead to a better understanding of parallelism and to the development of new models for their exploitation. Experimental machines will be a good tool for these investigations. In this article some of the basic issues in computer architecture are discussed using research proposals and existing machines as a guideline.

Introduction

In recent years the research activities in computer technology have increased considerably, in particular triggered by the impressive improvements in VLSI technology which allowed to develop powerful processors and memory chips. That the importance of this technology for the national economies is also realized by governments can be concluded from the number of projects that are undertaken on the national and sometimes international basis as in case of the European ESPRIT program. Almost all projects cover the following research topics:

- VLSI technology in particular submicron as well as 3D VLSI
- Software Technology in particular improvement of software productivity
- Supercomputers in particular pipelined vector computers
- Computer architecture in particular non-Von Neumann and parallel machines

Although all these items are of great importance for the improvements in computing, this review will concentrate on the last one.

A large number of computer architecture projects have in common that they are researching parallel systems. In such systems the management of the parallelism is the main problem. Therefore, this will be a central issue in this article which will also give some of the solutions chosen towards solving this problem.

Symbolic versus Numerical

In recent years a number of activities have been started to develop new computer architectures. These were triggered by the need for more computing power and the availability of cheap and
powerful VLSI based technology. The type of computer use can be divided into two quite distinct categories:

- numeric processing
- symbolic processing

One can characterize the difference between these two types according to the usage of the three basic architectural building blocks:

- CPU,
- memory,
- communication channel(s).

**Numerical processing**

Characteristic for this type of processing is the demand for optimal arithmetic operations preferably with floating point. The memory usage is characterized by predictable memory references because the type of data structures that are manipulated such as arrays and vectors are stored in the form of linear patterns. Because of the predictable memory usage the communication pattern is structured and often allows block type of transfers.

**Symbolic processing**

For symbolic processing logic operations (and, or, eor) are the predominant factor whereas arithmetic operations such as multiply and divide are rare. Moreover emphasis is more on such control flow operations as procedure calls demanding context switching. The main data structures to be handled are strings and have to be stored via linked pointer structures scattered all over memory, with as consequence that they are not predictable. The communication patterns between CPU and memory are far less predictable than in the previous situation.

Traditionally the focus in computing has always been on numeric processing, however, recent years have shown a shift in interest because research in Artificial Intelligence and Knowledge Based systems resulted in a growing interest in the symbolic processing. The most striking example is the Japanese Fifth Generation Computer System (FGCS) project [Mot82] where the emphasis is on the development of Knowledge Based techniques.

The impact of the FGCS project has been that a large number of research efforts in the area of Knowledge Engineering have been started all over the world. The type of machines used basically consist of an inference engine and a data base machine mostly of the relational type.

The research in inference and database machines was further stimulated by the possibilities of modern VLSI design.

However, also in the area of numeric processing we can observe dramatic changes caused by the impact of the same VLSI technology. Special processors for graphics and image processing are examples but also the modern micro processors which now have a computing power greater than a VAX computer, illustrate the leap forward coming from VLSI technology in recent years.
It can be expected that the impact of this technology will continue to increase the speed and functionality of future generation computer systems, however, a real big step forwards can only be expected if it becomes possible to exploit parallelism at all levels of computer architecture.

The speed of uniprocessor systems can be improved by adding parallelism for instance in the form of pipelining. Adding more and more parallel operations ultimately leads to systems with more then one processor the so-called multiprocessor systems. Fully utilizing the power of these system for numerical as well as symbolic computations demands understanding of parallelism on the level of programming techniques as well as computer architecture. This type of research will have to produce new concepts that will enable us to fully exploit the possibilities of the VLSI technology in the future.

Research in architecture

The research activities in architecture comprise such areas as computational models, software aspect of these models, the hardware organization and the possibilities and limitations set by VLSI technology.

To be able to describe the various levels of a computer system a certain amount of abstraction is used. In this abstraction the computer hardware and software organization and realization is pictured as a number of levels as illustrated in fig. 1 and fig. 2 [Bel78, Tan84]. In fig. 1 the point of view of a hardware designer is choosen whereas fig. 2 illustrates the software point of view. The ideas behind the layered model are that each of the levels has its own typical problems that demands the experience of particular specialists. The operating system level as an example demands system programmers to solve problems of file I/O or interrupt handling, whereas the micro programmer has to deal with completely other problems such as how to get the most horsepower out of his machine. For computer architecture the lower two levels of fig. 2 and the upper two levels of fig. 1 are the most important. The three lower levels of fig. 1 are the typical domain of the VLSI designer. The operating system level is the area of the system designer, whereas the high level languages compilers are the domain of compiler writers.

The discussion that will be presented here will focus on the problem of parallelism in architecture and the methods by which parallelism is exploited. Consequently we will classify computer systems according to the amount of parallelism present. Doing so we can follow the classification proposed by Flynn [Fly66] and distinguish systems according to the parallelism presented in control and data stream of the computer. This results in four catagories:

- Single Instruction stream Single Data stream (SISD)
- Multiple Instruction stream Single Data stream (MISD)
- Single Instruction stream Multiple Data stream (SIMD)
- Multiple Instruction stream Multiple Data stream (MIMD)
Systems applying the first two categories of parallelism are often called uniprocessors whereas the last two categories are called parallel processor systems.

First the trends in the developments of uniprocessor architectures will be given, thereafter parallel processor systems will be discussed.

Most of the projects that will be used as examples are in a research phase, however, also existing commercial computers will be taken into consideration.
Uniprocessors

As was discussed before the research in computer architecture is trying to find a suitable computer model and thereafter realizing the correct implementation. In the design of such a model the software aspects thereof are playing an increasingly important role. In the realization phase the hardware organization as well as the VLSI implementation aspects are setting important boundary conditions.

Models and languages

For the classification of the architecture the model on which the control and data exchange mechanisms is based is the most important parameter.
Specially the control mechanism has an important impact on the architecture. Traditionally there has always been a strong link between the type of control mechanism used in the architecture and the language applied to program the computer. It has to be stressed here that this resulted from the ideas that a particular control mechanism was better to support one class of languages whereas another was better for another class. These ideas were never really checked on their validity.
It is in no way obvious that a data flow mechanism is the best suitable model for executing a special language designed for data flow programming like VALID (ARVIND), whereas the language like FORTRAN which was designed to be executed on control flow based computers could run equally efficient on a data flow computer. The only argument that can be given at this moment in time is that control flow (von Neumann) type of computers have the longest history for sequential programming and are providing the most efficient executing environment because there exists the longest experience on how to optimize their performance.
Before discussing advantages and disadvantages in further details we will first look in the models in more details.

If we follow the point of view expressed in figure 2 a high level language can be seen as a virtual machine which runs on top of another virtual machine. The operating system level machine runs on top of the assembly level virtual machine which on its turn is running on top of the micro program level or is directly being executed by a hardware interpreter. The lowest level interpreter can use one of the three following mechanisms for control:

- Control flow
- Data flow
- Demand flow (Reduction)

In the control flow architecture the sequence of instructions, a program is composed of, is executed in a predetermined order. This is either sequential or parallel, but in any case in a firing sequence determined by the program counter. It is possible to optimize the von Neumann control flow model by exploiting so-called pipe lined parallelism. The languages that traditionally are designed to run on computers using this type of control mechanism are procedural in nature such as Fortran, Pascal, etc.
In a data flow architecture the sequence in which the instructions are executed is triggered by presence of the required arguments (data items). A program is decomposed in a flow graph with instructions on the nodes and data tokens flowing over the arcs (see fig 3). If the required arguments for an instruction are presented on the input arcs the result becomes available on its output arc. The type of languages used to program this type of machines are the so-called single assignment languages like ID, VAL, VALID, etc.

In the demand flow model the sequence of instruction execution is controlled by the demand for results. Similar to the data flow mechanism a program is decomposed in a flow graph, however, the execution of a particular instruction is triggered only when its result is requested.

The request for a result on its turn triggers request for arguments on the input arcs (see fig. 4) which might trigger other request for results from preceeding instructions etc. Languages used are the so-called zero assignment or applicative languages like FP (Backus), SASL, MIRANDA (Turner) and pure LISP.

There are architectures which are using more complex control mechanisms as the ones described above. Examples are the logic and actor architectures. These can be considered as extensions of the models discussed up till now but will be using more complex control mechanisms. The consequence is that their machine instructions are closer to those applied in high level languages.

In the logic architectures a certain goal is unified with particular definitions. When the goal is requested a search for the definitions or subgoals necessary to obtain the result is triggered which is similar to the situation in demand flow. The type of languages eg Prolog (Colmerauer) or derivatives thereof are based on first order predicate logic.

In the actor architecture a certain instruction or combination of instructions is triggered when a particular control pattern is presented. Examples of languages used for this concept are Small-Talk (Ingalls) and Pool (Philips).
Hardware aspects

During the realization of an architecture the following type of problems have to be solved:

- how can processor speed be optimized,
- how should the processor be organized,
- what type of memory organization should be chosen.

The first two issues are dealing with processor performance whereas the last one deals with adapting the processor with the memory speed. More performance can be realized by pipelining. Adapting of processor to memory speed can be obtained using cashing techniques.

Pipelining

Pipelining is a technique which tries to reduce the idle time of each of the parts of the CPU. The parts are constructed in such a manner that each of them can operate without any dependency on any of the other. It than becomes possible that while the ALU is busy executing the current instruction, the control unit is at the same time busy decoding the next instruction and the external bus control is busy fetching the third instruction. This concept results in a logical organization of the functional units in what can be regarded as a pipeline - hence the name - which will in reality be of a length determined by the number of units actually operating independently. A problem reduces the speed improvement resulting from this technique is the fact that conditional instruction will break-up the pipeline.

Cache memory

A cache memory is a special purpose intelligent memory device that is located between the CPU and the external memory. It saves the latest n - where n is the size of the cache memory - references to the external memory. Each time the CPU references a specific memory location, the cache memory may have a copy of the specified location. If that is the case, the cache memory provides the CPU with the data and the external memory is not accessed. If the cache does not have a copy, the external memory is addressed and the data is delivered to the CPU. In this situation the memory also stores a copy so the external memory need not be referenced next time the data is required. Like the usage of registers, the application of cache memories, may reduce the number of references to external memory. This works because theoretically a program, during certain parts of its execution, references a rather small set of memory locations, the so-called working set.

Three types of cache memories exist:

- Instruction caches
- Complete caches
- Address caches, sometimes called Translation Lookaside Buffers (TLB)

Instruction caches store instruction references only. Because instructions should never change during execution of programs, there can never be a discrepancy between the copy of a memory word that is stored in the cache and the actual contents of the corresponding memory location.
Discrepancy can exist for data, causing the cache to have a false copy of the corresponding memory location. This might occur in complete caches that can contain instructions as well as data. If the CPU writes results back in such a cache this cache will have a different copy of the data as stored in the main memory. This is known as the "data coherence problem". This problem is even more severe in MIMD systems where we have to deal with several CPU's working together each updating its local cache.

During the transformation of virtual addresses to physical addresses several tables are referenced such as page and segment tables. These are normally placed in RAM. This implies that translation of a memory address requires several memory accesses, which may take quite some time. Application of an address cache memory which caches a number of virtual/physical address combinations may significantly improve performance.

A disadvantage of cache memories is the complexity needed for solving the data coherency problem. Other factors which often complicate the design and application of cache memories are the need for high speed and the relatively large chip surface area such caches consume.

Enhancement of instruction set

Another source for improving CPU performance is enhancement of the instruction set. This is important for the CPU's of general purpose computers the so-called Complex Instruction Set Computers (CISC). Most modern processors which are at the core of these computers are designed with the idea in mind that increasingly powerful instruction sets automatically lead to better performance. Reasons that support this idea are:

- More functions in hardware imply faster execution
- Since the instruction set matches high level language constructs, it is easier for compilers to produce efficient code.

Enhancing the power of an instruction set can be done by adding instructions for string operations, floating point operations, complicated addressing modes - easy code generation for array references - index check instructions, frame construction instructions etc.

In addition most modern CISC's also have facilities for multiprocessing like fast process switch capabilities.

The control logic required to implement these type of instructions is very complicated. The level of complexity is such that it is nearly impossible to implement these instructions entirely in hardware. To enable implementation an extra interpretation level is added. This level is called the "micro-programming" level. The micro program consists of very simple instructions that can be implemented in hardware. The micro program on its turn interprets the conventional CPU (assembly level) instructions. This design philosophy is applied for most modern Complex Instruction Set Computers like micro-processor based systems.
Reduced Instruction Set Computer (RISC) [Pat85]

One of the greatest advantages of micro-programmed processors is that micro-programs are just programs and can more easily be changed than digital logic (the first requires a new ROM mask, the second a completely new chip mask). Also, the addition of customized instructions is possible without exorbitant costs. However, there are a number of disadvantages of micro programming too:

- Although debugging of micro programs is more easily done than debugging of corresponding digital logic, the micro programs for general purpose processors are of such enormous size (several hundreds or thousands of instructions) that errors are unavoidable, even in the final product.
- Compilers often use only a subset of the complete repertoire of instruction available to them. The unused instructions have been unnecessarily implemented.
- In complex micro programs pipelining is more difficult to implement.
- Decoding of instructions from a large set generally takes longer than decoding instructions from smaller instruction sets. This extensive decoding is slowing down machine performance, this because it has to take place at each microcycle. Often the addition of one new instruction or instruction facility would result in a far more complex decoding for the microcode (n+1 effect).

Also the VLSI designers have a problem with the complex decoding required for large microprograms. VLSI technology allows far larger chip densities on regular patterns such as memories and register files then for random logic as is required for decoding. As the silicon area can only be used ones there is a trade-off between more registers or larger ALU's or a cache and complex decoding hardware.

All these arguments have led to a trend in computer architecture to move away from the ever more complex instruction sets of CISC's and their microcode implementation approach and use the chip area to produce faster and simpler computers based on hard wired decoding the so-called Reduced Instruction Set Computers (RISC).

CISC's versus RISC's

The most important issue in the CISC - RISC controversy is whether complex instructions really speed up the use of high level language compilers or with other words are complex instructions really filling the semantic gap and is that all we want. The basic arguments of the RISC advocates is that compiler techniques have become mature and that the design of a processor and in particular the decisions what to put in the instruction set has to become a more scientific process. If there is a need to implement a new instruction the trade-off has to be made between the cost of making the instruction out of simpler instructions by defining a software construct or choosing to implement it in hardware.

These and other arguments have led to the following working definition for a RISC architecture:

- Instruction size is fixed.
• Instructions execute in a single (pipeline) cycle. Because the instruction size is fixed it permits simple decoding.
• Instructions are based on the Load - Store model eg is register oriented. The Load-Store model prescribes that each operation that has to store results must use a register to do so. The only transfer between data path and memory that is taking place has to go through the intermediate stage of a register.
• Only a limited amount of instructions are supported. If more complex instructions are necessary the compiler will built them out of simpler ones.
• Static runtime complexity is moved to compile-time.
• Hardwired control.

Disadvantage of RISC architectures is that programs increase in length. More instructions per unit of time are executed, with as a result, more references to memory. This in turn requires a high processor memory bandwidth and cashing. It has also been shown that RISC architectures have only average floating point performance as a result of the fact that these operations must be entirely coded in software.

Research items specific for parallel architectures

In this section those aspects of architecture which are specific for the design of parallel systems will be discussed. Some of these points will have more impact on the architectural models or their software aspect others are more important in the realization or implementation phase. However, most of them play a role throughout all the phases of architecture.

When designing parallel systems the following points have to be considered:

- system integration and cooperation
- communication and synchronization
- exploitation of parallelism
  - granularity of parallel operations
  - implicit versus explicit parallelism
  - distribution of processes over processors
- memory organization
- type of interconnection scheme (network)

System integration and cooperation

The research in system integration and cooperation deals with questions like whether all components in a system perform work which is part of one single task or perform independent tasks not requiring much interaction. In the first situation processes or processors might have to wait for results of others in order to proceed, which is not necessary in the second case. This has important consequences for the type and the amount of synchronisation applied by the computational model.
Communication and synchronization

Data exchange mechanisms can be exploited to realize communication and synchronisation. Two different data exchange mechanisms can be distinguished. One that relies on the presence of a shared memory and one which does not make this assumption: the so-called message passing concept.

A method for implementing communication and synchronization using global addressable (shared) memory is by defining a critical region in which access to shared data is controlled by a semaphore. A method for implementing communication and synchronization in a system without shared memory is message passing. It consists of transferring some data (a message) from one processor to another. These messages contain the address of the sender as well as that of the destination. When the sending processor and the destination processor are not adjacent, the processors in between sender and destination pass the message along in the right direction, which they infer from the destination address.

There are languages like ADA that have communication and synchronization primitives. Using mail boxes together with the ADA 'rendez-vous' mechanism it becomes possible to let one task communicate and synchronize with another task. The ADA rendez-vous mechanism is active in nature. Two processes agree via the rendez-vous mechanism to communicate and synchronize.

The critical region is an area in shared memory protected by a semaphore. The critical region concept is a passive way of communication and synchronization. A process has to request permission to enter the critical region.
These are examples of high level communication and synchronization, primitives which give rise to significant software overhead. A way to reduce this overhead is to implement special hardware for these tasks.

Exploitation of parallelism

The size of the smallest operation that is considered as an indivisible unity during parallel operations is an important quantity. This is often called the granularity of the parallelism. In case the various instructions of a control flow computer are used as smallest operations it is called 'fine grain'. In case parallelism is exploited at the level of procedure calls it is called 'medium grain' whereas the level of a full task or program is called 'coarse grain'.

Important in the design of an architecture is that the amount of communication is roughly reversely proportional with the size of the granularity in the parallelism. This implies that a fine grain system requires a large amount of communication between processes and processing elements, whereas a coarse grain system does not require much communication overhead.

The decomposition of problems in subtasks that can execute in parallel is an important issue. Often three types of parallelism are distinguished: explicit, implicit and pipeline parallelism.
The first two types will have consequences for the high level languages whereas the last one, that was already discussed before, only has consequences for the hardware.
Examples of implicit parallelism are the AND and OR parallelism in Prolog and the automatic vectorization in some Fortran compilers.
Explicit parallelism in a language allows a user to define tasks that can be executed in parallel. The distribution of tasks or processes over the available processors is an activity that either can be done at compile time, at run-time, or both. The first situation it is called static the latter dynamic binding. Sometimes a third binding method is distinguished in situations where one fixed algorithm is implemented in hardware. This is called bind at design time.

In situations where one is dealing with binding at compile and execution time the uniform distribution of tasks over processing elements becomes important. This procedure is called load balancing and is particularly complicated when dealing with a bind at run time system because one needs a method that can dynamically allocate processes. In such a system the type and speed of the switch connecting the processors with the memory system the interconnection network is of considerable importance.

Memory organization

Memory organization is an important factor in designing an architecture. Especially where the aim is to develop a system where a large number of processors (>100) need access to one shared memory system hierarchical structures are of interest. Such memories are realized as a number of levels with decreasing access time, where the fastest memory will be directly accessible by the processors. This is another form of caching because the whole memory structure will have a fast average access time without the requirement of building it all from fast and expensive components.

Network

Fig. 5 illustrates a number of possibilities ranging from expensive but fast networks like the crossbar switch to cheap but slower ones like the shuffle exchange networks, they will be discussed in more detail later in this paper.

Figure 5. Network configurations
Parallel systems

In this section the characteristics of SIMD and MIMD systems will be discussed, using a number of computer architecture projects as illustrations. When discussing case studies in parallel architecture only those parameters will taken into consideration which are of relevance to the particular project.

SIMD architectures

In SIMD architectures one control unit conducts a number of data-paths in parallel. Each of the data-paths has its dedicated channel to a memory element in which the data to be processed is stored. The fact that only one control unit is involved implies that all data-paths perform the same type of operation in parallel. By this virtue this architecture is in particular powerful for situations where the same type of operation has to be performed on a large amount of data. Typical applications thereof can be found in the signal processing, image processing and graphics processing field.

Examples of SIMD architectures

Examples of the application of SIMD architecture are the so called "Cellular Array Processors" (CAP). Today chips can be bought on which 72 1-bit processors (in a 8 * 9 array), each equipped with 128 bits of memory. The processors are just ALU's, with some simple bit operations. In addition they are able to communicate data to each of their neighbours via attached communication lines. These processors do not have jump instructions. The instructions to execute are not fetched by each processor on it's own, but are sent to each processor simultaneously by an external controller. To be able to proces data, this data must first be read into the processor array. This is done by sequentially feeding the data to one processor, which shifts it to it's neighbour etc. After processing, the data is exported in much the same way. The only conditional instructions that are possible are realized, in a simple way, based on condition code bits used by the external controller.

A promising design can be found with the ITT CAP-II chip [Mor85]. The first implementation should be available in 1987. This chip will incorporate 16 processors in a 4 * 4 array, each using 16 bit words. The instruction set is RISC-like, but instructions for floating point calculations are provided. The processors are reconfigurable to a word size of: 32, 48, 64 or 256 bits. Each processor has some condition code bits which determine whether the 'then' or 'else' part of an instruction is executed. Each instruction consists of these two parts. During program execution, one or more processors are halted during the 'then' phase, while others are halted during the 'else' phase.

The performance of one ITT CAP chip is estimated to be equal to two thirds of that of a Cray-1 super computer for those cases where optimal use of its features can be made.
It is possible to transfer some of the centralized control of the SIMD architecture to the individual processors but nevertheless maintain centralized control or better synchronization. In that case the boundary between SIMD and MIMD becomes confuse. An example of a machine based on this principle is the connection machine [Hil85]. The processing element is a one bit processor with a limited functionality. The centralized control prescribes when the next instruction has to be executed, however, locally a processor can decide whether to skip this execution this based on conditions present in its flag register. The connection machine can be equipped with 16 or 64K processors and uses a direct connection network for processor interconnection. These properties classifies the machine as an Ultra computer type which will be discussed in the next sections.

A step closer to a MIMD system is a so-called systolic array. This architecture is useful for those computations that have a very regular structure. A systolic array consists of one row or a complete array of simple processors each performing a part of a total computation. Data is shifted through the array in a predefined way such that the result is that the total array performs the overall computation. The most well known form of systolic processing is matrix multiplication. One part of the matrix is shifted in from left to right over the various rows whereas the different coefficients are stored in the processing cells. The results of the individual multiplications are shifted down in the columns and the results come out at the bottom of the array.

MIMD machines

In MIMD architectures, a (possibly large) number of processors, each executing it's own program, are working together on the same problem. An example of such a problem is that of sorting data. Each of the processors takes a part of the data thus sorting proceeds in parallel. After all processors have finished their jobs the separated sorted list must only be merged, which is a relatively easy operation.

To make it possible that several processors are working together on one problem, the data involved must be accessible by all processors. Moreover it is necessary that all processors can communicate with each other. Depending on the bandwidth of the communication channels, we speak of two types of MIMD architectures:

- Loosely coupled systems or networks (low bandwidth).
- Strongly coupled systems (high bandwidth).

Loosely coupled systems

In loosely coupled systems, processors are coupled by communication lines of low bandwidth (often serial connections). To obtain reasonable performance, each processor is equipped with a local memory. A single combination of processor and local memory is called a node. A node notifies another node that it needs certain data by using message passing communication over the serial or parallel link between the processors. We will not discuss these systems here any further.
**Strongly coupled systems**

In strongly coupled systems, processors are interconnected by high bandwidth communication channels. The channels are often parallel. Due to the high communication bandwidth, it is no longer an absolute requirement that each processor is equipped with its own local memory, however, in most cases it will be there for performance reasons. A fast parallel switch is used to connect processors and memories with each other.

The simplest form of such a switch is a bus architecture [Bor85]. It is an example of an indirect connection network. These are mostly used in industrial applications. When designed properly, this architecture possesses some redundancy, which increases the reliability of the system. By means of a bus arbiter, each processor can gain control of the bus. Two widely applied bus standards are the so called VME-bus and the Multibus II. VME-bus, actively supported by Motorola, is an asynchronous bus with a transfer rate of 20 Mbyte/s. Multibus II, supported by Intel, is a synchronous bus operating at 40 Mbyte/s. A new standard is Future bus which is destined to become also an IEEE standard. The maximum communication speed will be 5 to 10 times higher than that of the VME of Multibus: almost 300 Mbyte/s. Similar to VME Future bus is an asynchronous bus system. More bandwidth can be realized using more complex communication paths.

In a crossbar switch any PE (PE is a combination of a processor with memory) can communicate with any memory M at any time (see fig. 5). This in contrast with an mesh type of structure where any PE can only communicate with its four neighbors. The disadvantage of the crossbar is that its size grows with $N^2$. $N$ being the number of PE/M pairs connected to the network.

A shuffle exchange network like the Omega or 'delta' or 'Banyan' network has as advantage over a crossbar switch that its size is only growing with $O(N\log N)$ however, at the cost of an average delay of $\log N$ cycles. In this network any PE can communicate with any M but under worst case conditions the delay can be considerably larger than average. A configuration applying this type of networking has to cope with the fact that the response time can never be guaranteed to be the same at all times.

Direct networks like the cube, the tree and the mesh only allow a PE/M element to communicate directly with a limited number of neighbors in the system. However, the properties are such that response can always be guaranteed within a certain delay time. When communication with others than neighbors is necessary extra relaying delays are involved dependant on the routing that was chosen.

With indirect connection networks memory can be divided into several separate banks that may be used in parallel. Some of these networks (Banyan, Benes rearrangeable network, double tree, delta network) are built from simple switching components. They differ in the amount of possible parallelism (important when two processors want to address the same memory bank), the amount of redundancy (important when a processor fails) and speed (how many switching elements must be passed on the path from processor to memory?).

**Examples of MIMD architectures**

In this section a case study of existing or proposed strongly coupled MIMD machines will be given. The main emphasis in most of the projects is on designing MIMD systems exploiting as
UNDERLYING PRINCIPLES OF EXPERT SYSTEMS

Christian R. Pellegrini
University of Geneva, Geneva, Switzerland

ABSTRACT

Expert systems are also called knowledge-based systems. Thus the knowledge representation scheme constitutes one of the most important principles used in the design of an expert system. The second important principle is search. These underlying principles in addition to several others are presented, discussed and illustrated with some examples.

I. INTRODUCTION

In their 1975 Turing Award lecture Allen Newell and Herbert Simon [1] emphasized two basic concepts: symbols and search. They were mainly concerned with "physical symbol systems" that manipulate collections of symbolic structures and perform problem-solving tasks using heuristic search. Since expert systems are intended to embody the knowledge and intelligence of experts, the concepts of symbols and search are central.

Symbol structures can be used to represent knowledge in an expert system and different search strategies are commonly used to infer new knowledge from existing one.

In the following chapters several knowledge representation methods are discussed and the most important approaches to inference are compared for their effectiveness.

The goal of Artificial Intelligence (AI) scientists had always been to develop computer programs that solve problems in a way that would be considered intelligent if done by a human. Expert systems are the fruit of a 20-years quest to define the appropriate nature of such programs.

The development of AI research can be divided into three phases:

1) Find general methods for problem-solving and use them to create general-purpose program;

2) Find general methods to improve representation and search and use them to create specialized programs;

3) Use extensive, high-quality, specific knowledge about some narrow problem area to create very specialized programs.

The last phase produced what is called now expert systems. The process of building an expert system makes use of many basic concepts of AI, the most important ones will be presented in the following sections.
II. SYMBOLS AND INTERNAL REPRESENTATIONS

An internal representation is a stylized version of the world. The same representation may be embodied in a variety of different data structures, to make different operations efficient. But we will assume that they are all variants of the same abstract internal representation. For expert systems the internal representation is mainly used to embody the knowledge. A knowledge representation is a combination of data structures and interpretive procedures that, if used in the right way in a program, will lead to a "knowledgeable" behaviour. Work on knowledge representation in AI has involved the design of several classes of data structures for storing information in a computer program, as well as the development of procedures that allow "intelligent" manipulation of these data structures to make deductions, which can be defined as "logically correct inferences". This means that deduction from true premisses is guaranteed to result in a true conclusion. As shown by figure 1 the contents of the knowledge base is made of two types of information: facts and rules. The facts are the assertions that are not expressed as implications. Typically, they represent specific knowledge relevant to a particular case. The rules provide a formal way of representing recommendations, directives or strategies. They express general knowledge about a particular subject area and are given in implicative form.

![Diagram of Knowledge Base and Inference Engine]

**Figure 1: Structure of an Expert-System**

All the representation schemes are not suited for knowledge representation. Among the most widely used we will discuss:

- predicate calculus,
- semantic networks,
- frames.

II.1 Predicate calculus

Logic was one of the first representation schemes used in AI. It is important by two different aspects. The first is consideration of what can be said, the second is the deductive structure, the rules of inference.

The most fundamental notion in logic is that of truth. A properly formed statement, or proposition, has one of two different truth values, TRUE or FALSE.

Typical propositions are:

- John's car is red
- Seven plus five equals ten
- Bob is Mary's uncle.
The value (TRUE or FALSE) assigned to a proposition does not depend on
the meaning of the sentence. For example we do not verify if Bob is the
brother of one of Mary's parents to assign a value to the proposition c)
given above.

Pure, disjoint proposition are not very useful. The introduction of
connectives allows the combination of simple propositions into more complex
ones.

They are four important connectives: and, or, not, if.

The following table gives the truth of a combination in terms of the
truth of the connected propositions p and q.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p and q</th>
<th>p or q</th>
<th>not p</th>
<th>if p q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Note: in propositional calculus "if" does not express causality.

To deduce new sentences from previously given sentences, we use what is
called an inference rule. The best known inference rule is modus ponens. It
states that if we know that two propositions of the form X and (if X Y) are
true then we can infer that the proposition Y is true.

More formally, the modus ponens rule is expressed as:

(if (X and (if X Y)) Y)

We can think of the modus ponens rule as the "if-elimination" rule,
because it is possible to replace X and (if X Y) by the single statement Y.

For the purposes of AI, propositional calculus as described till now is
not very useful. In order to capture more knowledge, it is not only
necessary to be able to express true or false propositions, but also to be
able to speak about objects, to specify relationships between objects and to
generalize these relationships over classes of objects. We turn to predicate

We can define predicate calculus as:

\[
\text{predicate calculus} = \text{propositional calculus} + \text{predicates + quantifiers}
\]

Predicates

We call predicates, statements about objects and in relation to other
objects. A predicate is applied to a specific number of arguments and has a
value of either TRUE or FALSE.

Example:

A one-argument predicate: (is-red X)

when applied to "the queen of hearts", the predicate has the value TRUE

(is-red the-queen-of-hearts) is TRUE

when applied to "this sheet of paper", the predicate has the value FALSE

(is-red this-sheet-of-paper) is FALSE.
Quantifiers

We shall often have occasion to refer to facts that we know to be true for all or some objects of a class. For this we introduce two notions, those of variables and quantifiers. A variable is a name in a statement that must be replaced by a constant in order to be able to evaluate the statement to TRUE or FALSE, such as "X" is the predicate "is-red".

There are two kinds of quantifiers, existential quantifier and universal quantifier. Universal quantifier says that something is true for all possible values of a variable. We indicate that X is a universally quantified variable in the formula f with this :

\[
\begin{align*}
\text{(forall } (X) \ f(X)) & \quad \text{explicit quantifier} \\
\text{(f(?X))} & \quad \text{implicit quantifier}
\end{align*}
\]

Example :

Using universal quantification we can express the statement that "all elephants are grey" by :

\[
\text{(if (elephant ?Z) (color ?Z grey))}
\]

This states that for all objects, if the object is an elephant, then the color of the object is grey. The translation of the English sentence into a predicate is carried out using the if connective.

The other kind of quantification is existential quantification. We indicate that a variable is existentially quantified like this :

\[
\begin{align*}
\text{(exists } (X) \ f) & \quad \text{explicit quantifier} \\
\text{(f(X))} & \quad \text{implicit quantifier}
\end{align*}
\]

Example :

To say that every person has a head we would say :

\[
\text{(if (person ?X) (exists (Y) (head-of ?X Y)))}
\]
or more simply by dropping the explicit quantifier "exists (Y)"

\[
\text{(if (person ?X) (head-of ?X Y))}.
\]

With the introduction of quantifiers, we have also introduced a new rule of inference, called the universal instantiation (or universal specialisation), which states that :

if something is true for everything, then it is true for any particular thing

or more formally :

\[
\text{if we have : (forall X P(X)) \quad P : any predicate}
\]

we can conclude : P(A).

This rule of inference is also called the forall-elimination rule.

Example :

if we know that (if (man ?X) (mortal ?X))
we can apply this to the individual "Socrates" using the universal instantiation to get

\[(\text{if (man Socrates) (mortal Socrates)})\].

Given the rules of inference "modus ponens" and "universal instantiation", one can infer new facts using the standard techniques of mathematical proofs. One starts with already established information and deduces new information until one has deduced the fact one wanted.

Example:

Deduction of "Tweety is yellow" given that "Tweety is a canary" and "all canaries are yellow"

\[
(\text{canary tweety}) \quad \text{assumption 1}
\]

\[
(\text{if (canary ?X) (color ?X yellow)}) \quad \text{assumption 2}
\]

using universal instantiation on assumption 2 we produce:

\[
(\text{if (canary tweety) (color tweety yellow)})
\]

using modus ponens on the above instantiation and assumption 1, we get:

\[
(\text{color tweety yellow}).
\]

II.2 Knowledge Representation Using Semantic Networks

The term semantic network, or semantic net, is used to describe a knowledge representation method based on a network structure. Semantic networks, developed originally for use in cognitive psychology, are now a standard representation method for AI and expert systems. A semantic network consists of points called nodes connected by links called arcs describing the relations between the nodes. Both nodes and arcs can have labels. Nodes usually represent objects, concepts or situations in the domain of knowledge. Arcs can be defined in a variety of ways, depending on the kind of knowledge being represented. Common arcs used for representing hierarchies include isa and has-part. Semantic nets used to describe natural language use arcs such as, agent, object and recipient. The figure 2 shows the structure of a semantic net.

![Figure 2: Structure of a Semantic Net](image-url)
As a simple example, consider the following sentences:

- "The Queen Mary is an ocean-liner"
- "Every ocean-liner is a ship".

These can be represented in a semantic net form as:

```
Queen
Mary isa Ocean
Liner isa Ship
```

Because we know about the properties of the relations linking the nodes (e.g. the isa relation is transitive), we can infer a third statement from the net shown above, that "The Queen Mary is a ship", even though it was not explicitly stated.

The isa relation and others like the has-part relation establish a property inheritance hierarchy in the net. This means that items lower in the net can inherit properties from items higher up in the net. This saves space since information about similar nodes doesn't have to be repeated at each node. Instead it can be stored in one central location as shown in figure 3:

```
SHIP
    isa
  OCEAN
Liner has-part
    isa
  OIL
TANKER
    isa
  ENGINE
    has-part
  HULL
      isa
    QUEEN
MARY
  isa
  LIVERPOOL
    has-part
  BOILER
```

Figure 3: Semantic Net of the Concept of a Ship

The net can be searched, using knowledge about the meaning of the relations in the arcs, to establish facts like "The Queen Mary has a boiler".

In a domain where much of the reasoning is based on a well-defined taxonomy a semantic network is a natural representation scheme.

The use of semantic network to produce inferences by following the links must be done very carefully. Because some of the links may not have the transitive property needed. Consider the following semantic net:

270
If we add the following assertions:
- "The robin is an endangered species"
- "Endangered species are studied by naturalists"

we would create the following structure:

The problem illustrated by this simple example involves inheritance. Since the reasoning procedures treat the isa link as a property inheritance link, the node "Clyde" inherits all the properties of "Endangered species" just as it inherits the property of having wings from the "bird" node. In this way, one might conclude that "Naturalists study Clyde" which may or may not be true. The source of the problem is that there is no distinction in the semantic net structure between an individual and a class of individuals. Furthermore, some things said about a class are meant to be true for all members of a class, like "Robins are birds", while some refer to the class itself like "Robins are an endangered species". Some recent research has explored various ways of making the semantics of network structure more precise, mainly by defining different property inheritance strategies based on the knowledge attached to a link.
Semantic nets are also used in natural language research. They are particularly suited to capture the meaning of a sentence:

Examples:

A) Sentence: "Bill gives Judy a gift."
Semantic net:

B) Sentence: "Bill told Laura that he gave Judy a gift."
Semantic net:

The semantic net representation is also very useful to point out similarities in the meaning of sentences that are closely related but have different structures.

Although the sentences of examples A) and B) above look different, their semantic nets look similar. In fact, semantic net A is completely contained in the semantic net B.
II.3 Knowledge Representation Using Frames

Marvin Minsky [2], who originated the concept of frame, describes it as follows:

"A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed".

A frame is organized much like a semantic net (they are often both considered as frame-based knowledge representation). A frame is a network of nodes and relations organized in a hierarchy, where topmost nodes represent general concept and lower nodes more specific instances of those concepts. In a frame system the concept of a written report could be organized as:

```
  report
  ↓    ↓
  isa  isa
  ↓    ↓
progress report
  ↓    ↓
  isa
  ↓
progress report #35
```

This looks just like a semantic net. But in a frame the concept at each node is defined by a collection of attributes and values of those attributes.

**Examples:**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>John</td>
</tr>
<tr>
<td>color</td>
<td>red</td>
</tr>
<tr>
<td>size</td>
<td>small</td>
</tr>
</tbody>
</table>

These attributes are called slots. Each slot can have procedures attached to it which are executed when the value of the slot is changed.

A frame consists of two parts:

a) the declarative structure, the way the representation of static facts is organized;

b) the dynamic aspect that can be attached to a slot, the procedures, to drive the reasoning or problem-solving behaviour of the system.

They are three useful types of procedures often attached to slots:
1. if-added procedure:
   executes when new information is placed in the slot

2. if-removed procedure:
   executes when information is deleted from the slot

3. if-needed procedure:
   executes when information is needed from the slot, but the slot is empty.

These attached procedures can control the assignment of information to the node, making sure that appropriate action is taken when values change.

To illustrate how a frame system operates, we represent the report hierarchy given above with slots, values and procedures attached. For simplicity we assume that some slots have default values.

Suppose that a manager of a company requests a progress report on a special project. He would, if the system allows, request quite naturally:

"I need a progress report on the XYZ project".

The interface program analyzes the sentence and constructs the following frame structure:

![Diagram of the frame structure]

Figure 4: A Frame Structure
The following procedures execute one after the other:

1) The if-added procedure attached to the "topic" slot executes because a value was inserted in that slot. It finds in the database the name of the project leader of the project XYZ and puts that name in the slot "author".

2) The if-added procedure attached to the slot "author" executes, because a value was inserted, and produces a message to send to the project leader. To do that the procedure needs the due date which isn't there, it activates the if-needed procedure attached to the slot "due date".

3) The if-needed procedure determines the due date according to today's date and inserts the value in the slot "due date".

4) The if-added procedure attached to the slot "author" finds that another information is needed; the length of the report. As the slot "length" has no attached procedure to supply the value, a search is made in the node above the node = 15, the one for the general concept of progress report and the value for the length is found. Then the procedure can produce the message to send to the project leader.

If at any time the value of the slot "author" is removed, the if-removed procedure will automatically send a message to the project leader telling him the report is canceled.

III. SEARCH AND DEDUCTION

In the preceding sections we have described the most frequently used knowledge representation schemes, we then turn our attention to the production of new knowledge elements from inputs. This is essentially done by "deductive" reasoning processes, in that they are concerned with drawing as many useful and probably true inferences as possible from the input.

An important abstraction is the notion of "knowledge-base", an organized set of structures that contains the facts and the rules which represent what the computer currently "believes". Between the knowledge-base and the program using it, there is a program responsible for adding or deleting beliefs, and performing certain types of inference. Some of these inferences are done when facts are added, some when requests for information arrive. This special program is called the inference engine.

The simplest design is to use the predicate calculus representation for the knowledge-base and an inference engine that contains no facts at all, just the basic rules of inference of predicate calculus. The inference engine has a choice when to do inference. It could do it as soon as a new fact is asserted, or it could wait until a query is received. For a predicate calculus system, assertion-time inference is called forward chaining and query-time inference is called backward chaining. The process is then to search the tree produced by the last technique for the appropriate solution.

III.1 Forward Chaining

The inference rule of modus ponens will not allow us to conclude anything from:

\[\text{a)} \quad \text{if (block ?X) (prism ?X)}\]
\[\text{b)} \quad \text{(block A)}\]

until we add one more rule:
from \((p(?X))\) infer \(p\) with all occurrences of each variable \(?X\) replaced by the same term.

Now the statement \((p(?X))\) allow us to infer

\[
(\text{if (block A) (prism A)})
\]

by replacing \(X\) with \(A\) and now modus ponens can get us to

\[
(\text{prism A})
\]

In order to avoid the inference of many other useless formulas, it is necessary to change the modus ponens rule to the following more general rule:

from \(p'\) and \((\text{if p q})\) infer \(q'\)

where \(p\) unifies with \(p'\) and the resulting substitution is applied to \(q\) to give \(q'\). This rule is called forward chaining. Unification means finding values of variables that make two expressions equal. The two expressions

\[
(\text{block ?X})
\]

and

\[
(\text{block A})
\]

can be unified by the variable value \(X = A\). In general the output of the unification process is a set of variable = value pairs, called a substitution. Each pair variable = value is called a variable binding; each variable is said to be bound in this substitution. Applying a substitution to a formula means replaing each occurrence of a bound variable with its value; in our case the result of applying the institution \(X = A\) to \((\text{prism ?X})\) gives \((\text{prism A})\).

**Notation:** we will use Greek letters for substitution such as:

\[
\Theta = \{ X = A \ , \ Y = \{\text{surface ?Z}\} \}
\]

With the notation, if \(p\) is a formula or term, \(p^{\Theta}\) is the result of making the institution of \(\Theta\) into \(p\). For example if \(p\) is

\[
(\text{touching (side ?X) ?Y})
\]

and \(\Theta\) is the substitution given above then \(p^{\Theta}\) is

\[
(\text{touching (side A) (surface ?Z)})
\]

Note that unification works even when patterns to be unified contains variables.

So far we have neglected an important issue. Suppose the inference engine is trying to forward chain on these two formulas:

\[
(\text{on ?X table})
\]

\[
(\text{if (on big-bertha ?X) (collapse ?X)})
\]

Intuitively, it should be able to conclude \((\text{collapse table})\), but there is no way to unify \((\text{on ?X table})\) with \((\text{on big-bertha ?X})\). The problem is that the occurrences of \(?X\) in both formulas are completely different variables. The solution is to rename all the variables in the two formulas that have the same name. This process is called standardizing the two formulas apart.
The figure 5 gives a graphical illustration of a forward chaining example.

![Graphical Illustration of Forward Chaining Example](image)

**Figure 5: An Example of Forward Chaining**

### III.2 Backward Chaining

In a typical deductive knowledge-base most inferences are done at query-time, using the rule called backward chaining. In this form of inference, the inference engine doesn't do anything with the implication (if p q) until if is trying to answer a query of the form q', when it proposes p' as a "subquery". Instead of query, we will use the term goal, because from the point of view of the deductive process the aim is to find instances of the query pattern, and such goal will have subgoals and sub-subgoals derived from backward chaining. In fact the goal-tree structure is the perfect structure to implement this process.

The formal definition of the backward chaining method is the following:

- original goal: (Show : q')
  - where (if p q) is in the knowledge-base
  - and q' and q have MGU
  - produce subgoal: (Show : p0)
  - and if the answer is γ, then $\mathcal{G} \cup \gamma$
  - is an answer to the original goal

**Notes:**

a) "MGU" means Most General Unifier, the most general set of variable bindings.

b) The term "Show" is used to distinguish between formulas that are asserted and formulas used as goals; the notation (Show : formula) is used to designate the latter.

Conjunctions must be handled specially in backward chaining. The goal (Show : r') will chain through (if (and p q) r) to generate the subgoal (Show : (and p' q')). This is called a conjunctive goal. This is to handled by finding variable bindings that are answers to both (Show : p') and (Show : q').
To illustrate this consider the problem of finding a proof for \((p ?x)\) given the rules:

\[
R_1 : (\text{if (and (q1 ?X) (q2 ?X)) (p ?X)}) \\
R_2 : (\text{if (and (r1 ?X) (r2 ?X)) (p ?X)})
\]

together with the facts:

\[
(q1 a) (q1 b) (q2 a) (q2 b) \\
(r1 c) (r2 d)
\]

This problem produces the following goal tree:

\[
(\text{Show: (p ?x)})
\]

\[
(\text{Show: (and(q1 ?x)(q2 ?x)))} \quad (\text{Show: (and(r1 ?x)(r2 ?x)))}
\]

\[
(\text{Show: (q1 ?x)})(\text{Show: (q2 ?x)}) \quad (\text{Show: (r1 ?x)})(\text{Show: (r2 ?x)})
\]

\[
x=a \quad x=b \quad x=a \quad x=b \quad x=c \quad x=d
\]

Once we have built the goal tree, the problem is to search it. The problem is that, without knowing more about the application domain, it can be difficult to find a solution to one child of an "and" node that is also a solution to all others. The problem is how to search through the space of partial solutions to a goal tree.

**Algorithm:**

1. The initial partial solution is "empty substitution, goals = top-goal";
2. A partial solution is extended by picking a goal \(g\), and finding all the assertions of knowledge-base of the form \(g'\) or (if \(p g'\));
3. If the assertion is of the form \(g'\), then the chosen goal is just eliminated from the set and the substitution is plugged into the remaining goal;
4. If the assertion is of the form (if \(p g'\)) then the conjuncts of \(p\) are added to the goal set as well.

At any moment the set of remaining goals is called a sibling goal set.

For example the sequence of partial solutions generated in the solving the tree given above is the following:
I
Use: Start
Order: 1
\( \emptyset: \{\} \)
Goals: \((p \ ?x)\)

II
Use: R1 on I
Order: 2
\( \emptyset: \{\} \)
Goals: \((q1 \ ?x)\)
\((q2 \ ?x)\)

III
Use: \((q2 \ a)\) on II
Order: 3
\( \emptyset: \{x=a\} \)
Goals: \((q1 \ a)\)

IV
Use: \((q1 \ a)\) on III
Order: 4
\( \emptyset: \{x=a\} \)
Goals: \(\{\}\)

Notes:

a) The term "Use" indicates how the partial solution was generated.

b) The term "Order" specifies the order in which the partial solutions are generated.

c) When an empty list of goals is reached, the substitution represents a complete solution.

What we really need is a heuristic evaluation function that will tell us when a set of goals is promising. The term "heuristic" indicates a processor a rule of thumb that may help in the solution of a problem, such as finding a solution in a large search space, but that does not guarantee the best solution, or indeed, any solution.

Consider the following knowledge-base:
1 (if( AND( above ?x ?y) (above ?y ?z))
   (above ?x ?z));
2 (if( on ?x ?y) (above ?x ?y))
3 (on bl-1 table)
4 (on bl-2 bl-1)
5 (on bl-3 bl-1)
6 (on bl-4 bl-2)
7 (above lamp1 table)
8 (color table beige)
9 (color bl-1 red)
10 (color bl-2 white)
11 (color bl-3 blue)
12 (color bl-4 green)
13 (color lamp1 yellow)

Consider the query : (Show : (and (above ?X table) (color ?X green))).
This query means : "Find something green above the table".

At any given time there are one or more solutions which could be extended. The heuristic used here is to work on the partial solution with the fewest sibling goals.

The figure 6 shows the search generated by the query given above and limited by our simple heuristic evaluation function.

Figure 6: Partial Solution for Backward Chaining
Discussion:
- the boxes labeled 16 and 17 represent the two alternative partial solutions derived from the partial solution in box 15;
- box 17 looks more attractive because it has only one sibling goal; but the attempt to work on box 17 fails immediately;
- looking back at box 16, we produce new solutions 18 and 19;
- box 19 is more attractive than 18 (since it has 2 siblings to 18's 3) and the evaluation function never finds box 18 attractive, so its "Order" slot remains marked "unused".

Note that the search space is surprisingly large even for such a simple problem. Fifteen partial solutions are generated, of which five are "blind alleys" (either they match nothing or they are post-poned for ever).

The figure shows only the derivation of one solution, but there may be others and partial solutions are left to extend. When should the inference engine keep looking? This is still a difficult question. If there is no variable in the original query, the process can stop after one answer. When variables are present, in some cases, one can do a precheck to find the number of possible answers and stop when that number is reached.

REFERENCES


ANNOTATED BIBLIOGRAPHY

This reference set discusses, among other topics: expert systems, natural language systems, various logics, knowledge-bases, search, graph representations etc.

Boden's book describes AI in a way that stresses its human relevance, particularly in relation to psychology and philosophy. Boden uses plain language and makes the understanding of difficult aspects of AI particularly clear.

This book is the product of a workshop intended to synthesize knowledge in expert systems field. It identifies the things on engineer or technical manager needs to know in order to undertake the implementation of an expert system. The books includes comparisons of various knowledge engineering tools and provide in-depth information on the different stages in implementation. Also noted are potential pitfalls and ways to avoid them.
Rich, E.  
"Artificial Intelligence".  
This is a highly readable introductory textbook on the techniques and implementation areas of AI. Rich provides clear explanation about knowledge representation methods, logic and reasoning, control strategies, heuristic search and other important concepts of AI.

Winston, P.H.  
"Artificial Intelligence" (2nd ed.).  
Addison-Wesley, 1984.  
Winston's classical textbook discusses basic theoretical concepts of AI, citing successful applications.

Winston, P.H. & Horn, P.  
"LISP" (2nd ed.).  
Addison-Wesley, 1984.  
This book gives the basics of LISP programming and demonstrates how LISP is used in actual practice. This second edition features COMMON LISP, the new standard for the language. This is a companion book of the book on AI.
DATA ACQUISITION SYSTEM DESIGN USING THE STRUCTURED ANALYSIS/STRUCTURED DESIGN (SASD) METHODOLOGY

J. Harvey

Rutherford Appleton Laboratory, United Kingdom

ABSTRACT

Traditional methods of developing software for High Energy physics applications need to be reviewed in the light of the increase in scale and complexity of modern experiments. The techniques used by software methodologies such as SASD provide a suitable framework for establishing a proper software design. Some of the benefits of taking this more rigorous approach to software design include improved reliability as well as improved maintainability. The deliverable items produced as a result of the design process provide an important way of communicating between members of the software team. In these lectures some of the principles of Software Engineering are reviewed and then an overview of the technical methods used by the Structured Analysis and Structured Design methodology (SASD) is given. Finally some of the automated tools being developed for enhancing the programmer's productivity in the design, coding and testing of software are mentioned.

1) INTRODUCTION

The current trend in high energy physics is towards the construction of very large colliding beam machines in order to produce higher collision energies. One consequence of this trend is that only a small number of experiments can be accommodated by the machines being limited by the number of regions where the opposing bunches are brought into collision. A second consequence is that the experiments themselves are extremely complex in that they attempt to investigate all interesting physics processes that occur during the collisions. These experiments are therefore designed to include a complex arrangement of detectors which cover as closely as is practicable the complete solid angle surrounding the interaction region. In addition the events themselves are extremely complicated and typically may include more than 100 secondary particles. Therefore as much information as possible is recorded from the detector to permit precise reconstruction of the events offline. The scale of the experiments is also reflected in the size of the collaborations participating in them (~400 people from ~25 institutes worldwide) and by their average lifetime (> 10 years).

To give an example, some of the parameters for the Aleph experiment being prepared for use on the Lep machine are:

- weight ~ 1500 tons
- volume ~ 1000 m$^3$
- readout ~ 700,000 electronic channels
- event size ~ 100 kbyte
- event rate ~ $10^7$ events per year

These features have serious implications for both the hardware and software needed in the data acquisition systems of these experiments. The hardware must cope with extra number of channels as well as the extra information per channel, such as pulse shape. This information must be processed to supress uninteresting data, to automatically apply calibration corrections, and to format the data to comply with data structure conventions. The large data volumes thus generated must be moved between processors and ultimately to the online data store. Fortunately hardware technology progresses rapidly. For example improved VLSI fabrication techniques have been exploited to design Flash ADCs, single chip microprocessors with a full 32 bit architecture,
Digital Signal Processors, customised chips etc. New improved data acquisition busses (such as Fastbus) and computer backplanes (such as VME and B1) have been developed and there is currently much progress being made in design of high capacity storage devices (such as high density cartridge tapes and optical disks). These developments have been eagerly taken up by the High Energy Physics community as is demonstrated by the fact that many of these topics are covered in detail in this School.

The complexity of the data acquisition systems also places major requirements on the software needed to operate the experiments. Firstly the use of distributed systems incorporating hundreds of microprocessors mean that more functions and decisions are now being entrusted to software. In addition the large volumes of data constants associated with calibration of the electronic channels, with describing the detector and with keeping track of bookkeeping information imply that sophisticated tools are required to organise and manage the online database. Facilities are also needed to manage access to all resources belonging to the data acquisition system to permit independent groups to work at the same time with the equipment without interfering with one another. The user interface should be easy to use and provide many automatic checking procedures. Given these requirements the question remains as to whether the traditional "ad-hoc" software development methods will suffice?

The aim of these lectures is to introduce some of the techniques which have been developed since the early seventies to provide a more rigorous framework for establishing a proper software design. In particular some of the principles of software engineering will be recalled and then an overview of the technical methods used by the Structured Analysis and Structured Design methodology (SASD) will be given. Finally some of the automated tools being developed for enhancing the programmer's productivity in the design, coding and testing of software will be reviewed.

2) HISTORICAL PERSPECTIVES

In the 1960s costs of software production were rapidly escalating and were outstripping the cost of computer hardware by more than a factor of three. In addition the indirect costs incurred were even bigger as software development was usually found to be the cause of delays in the completion of a project. This precipitated the so called Software Crisis. In an attempt to understand the situation studies of some major projects were carried out to try to see where the software effort goes1). One of the principle pieces of information discovered as a result of the study was a breakdown of the fraction of time spent in the different phases of a software project, ie.

- Analysis and design 30%
- Coding 20%
- Testing 50%

It can be seen that the time spent coding is surprisingly small whereas that spent testing is very high. The fact that once the coding phase is completed the project is only half completed explains why "software always appears to be late". The main conclusions drawn from this study were that firstly modularity of the program structure should be improved so that code could be more easily tested and also that it would be better suited to adaptation during its evolution. Secondly it was concluded that more effort should be placed in the early phases of the software development process since the most expensive errors to correct were found to be those traced to a poor understanding or misinterpretation of the requirements of the system.

Other problems concerning the management of large teams were identified2). It was found that large projects can sustain a build-up in manpower of ~30% a year and no more before the infrastructure of the team becomes strained. This provides evidence of the fact that the effort to communicate between members of a large team is great and that efficient ways of exchanging information are required. To facilitate this communication a convenient way of fragmenting a project into its components with well defined interfaces is essential. Another
related problem arises from the fact that turnover in the composition of team members must be anticipated and new people need to be accommodated into the project.

In response to the Software Crisis the earliest developments concerned a discipline for programming\textsuperscript{3}),\textsuperscript{4}). An analysis of errors introduced during the coding phase revealed that the majority of problems arose due to errors in the declaration of variables as well as faulty logic in those areas of the code concerned with flow of control. In particular it was found that the errors which persisted to the final stages of debugging and testing were dominated by errors in the flow control logic. Guidelines for coding were therefore introduced which were designed to maximise clarity and maintainability of groups of instructions by the use of mnemonic names, comments etc. Also the criteria for choosing program structure were not based on optimum performance but rather to avoid unrestricted control flow. This programming philosophy became known as \textit{Structured Programming} and new programming languages with additional redundancy were developed in an attempt to use the compiler to help in diagnosing program errors.

In the early seventies the concepts of Structured Programming were extended to include the organisation of program modules. A set of technical methods were developed for formulating the design of the architecture of the program in terms of a hierarchy of program modules. These methods together with a set of criteria for evaluating the quality of the design constitute the \textit{Structured Design} approach to software design. Thus Structured Design is more concerned with the overall structure of the program rather than the detailed logic of the individual procedures.

Both the above methods are implementation disciplines and do not address the problem of describing what the system being built should do and what the specific requirements are on its functionality and performance are. In the late seventies \textit{Structured Analysis} was developed\textsuperscript{7}),\textsuperscript{8}) which provides a method for building a rigorous requirements definition model to be used as input to the design phase. More recently the techniques of Structured Analysis have evolved to include control and timing requirements as these are essential for describing real-time applications\textsuperscript{9}). Moreover some of the limitations of the pure analysis and design methods have been overcome by the adoption of additional techniques, such as the State Transition Diagram for specifying control applications and the Entity-Relationship data model for describing the underlying structure of the data.

In some cases the originators of these techniques have packaged the technical methods with procedures for managing large software projects in an attempt to offer a complete support environment to aid in software design. In addition this effort has been complemented by the development of computer aided design packages (automated tools) which improve the efficiency with which the methods can be applied as well as providing checks for monitoring the integrity of the design.

3) PRINCIPLES OF SOFTWARE ENGINEERING

The approach to software design outlined in the previous section is commonly referred to as \textit{Software Engineering} which derives its name from the fact that it may be viewed as an attempt to bring an engineering-like discipline to the process of software design. In particular the manufacturing concept of 'life-cycle' has been used to describe all the phases in the life of a software system from the time it is first conceived until it is finally discarded. One of the fundamental principles of \textit{Software Engineering} is that extra effort put into the earlier phases of software development will be reflected in reduced costs during the testing and maintenance phases. One of the aims of this approach is to prevent errors from being introduced or at least to detect them at the earliest possible time. Various models exist for the life-cycle but by and large the phases may be summarised as follows:

- **ANALYSIS** - Functional descriptions of the systems intended behaviour are developed. The descriptions should include a statement of system functions, of external interfaces to the system's environment, of required data items and a list of any particular requirements on performance, design and implementation.
• DESIGN - The functional specification is implemented. Firstly the overall structure is defined in terms of the program modules and their interconnection. Then follows the detailed design in which algorithms are developed. Alternative system structures may be tried and particular detailed constraints applied.

• IMPLEMENTATION - This involves turning designs into code. The principles of Structured Programming should be adhered to as previously outlined.

• TESTING - This involves verifying that the system correctly performs those functions described in the functional specification.

• MAINTENANCE and EVOLUTION - This covers the repair, adaptation and enhancement that may be needed after the initial development.

Note that the boundaries between phases are not very strict and that it is often necessary to attempt an implementation in order to get information critical to design decisions (ie. prototyping).

The set of procedures followed through the life-cycle of program development constitute a Software Methodology. The purpose of the Software Methodology is both to indicate in more detail exactly what should be done during each phase in the life-cycle and also, by offering this set of techniques and tools, provide assistance on how to proceed at each stage. Modelling techniques are used to represent the problem in graphical or mathematical notation rather than in a textual form which tends to be less precise and often ambiguous. Management procedures are also used for tying the stages together and for ensuring that the technical methods can be made effective. In addition several companies offer automated tools which are themselves software products and which may be used for checking designs for completeness and consistency.

There are many Software Methodologies to choose from and a survey commissioned by the Ada Joint Program Office gives a good indication of each methodology's principle features and benefits\(^{10}\). At CERN SASD is now being quite widely used by teams from several experiments and also by groups working on the control systems for the SPS and Lep machines. SASD was selected because it covers all phases of the life-cycle, because it is widely used in industry (eg. Tektronix, Boeing, Lear-Siegler and ESA) and because it is supported by several sets of automated tools.

4) THE ANALYSIS PHASE

The goal of the analysis phase is to produce a functional specification of the requirements of the system. In addition to being graphic and concise the specification should also have other qualities. Firstly it should be correctly partitioned to show general views of the system as a whole as well as more detailed descriptions of the different, largely independent, components. In addition it should concentrate on what the system is meant to achieve rather than how the system should be implemented ie. should be logical rather than physical.

Structured Analysis uses a set of several different tools for modelling different aspects of the system. The Data Flow Diagram (DFD) is used to identify the principle data flows in the system and the processes which transform them into one another whereas the Entity Relationship diagram is used to describe the system's data structures and their interrelationships. More precise details on the nature of the transformations and of the composition of the data are specified in mini-specifications (using Structured English) and in the Data Dictionary (using a mathematical notation). Together, the output produced as a result of applying these techniques constitutes the functional specification. It should be emphasised that these methods are continually being refined as a result of experience gained using them. For example, attempts to apply these techniques in the design of avionics systems in industry\(^{9}\) has resulted in the introduction of new features, such as control and time dependency of data flows, which have proved to be important for the design of real-time systems.

This section contains an overview of these techniques; more detailed information can be found in the standard texts.\(^{11,12}\)
4.1 Modelling Transformations

The tool used for modelling transformations is the DFD. In its simplest form the DFD comprises four graphic elements:

- A labelled arrow represents a data flow
- A circle represents a process which is transforming its input data flows into its output flows
- A straight line represents a file or other storage element
- A square represents a source or a sink of data

Thus a DFD representing a simplified system for collecting and monitoring events (Figure 1) may be interpreted as follows:

"The Build Event process reads Digitisings from the Detector Readout System according to the Trigger Mask for that particular event and transforms them into Events. The Validate Trigger process then analyses Events and classifies them according to what type of interaction they correspond to. The Select Events for Logging process then reads the Classified Events and matches their classification against the Event Selection Mask. If there is a match the Classified Events are formatted into Event Records and written to the Online Data Store. At various stages in this dataflow pipeline the data may be sampled to Build Statistics of the detector performance which are stored in a Histogram File. Histograms may be sampled at any time by the Operator and sent to a Display monitor by the Display Statistics process."

Figure 1 Simple example showing principle elements of the Data Flow Diagram
This simple notation may be developed to describe the system's dynamic character by extending the notion of dataflow to include time. Time continuous dataflows, i.e., flows that exist at every instant within a time interval, are useful for representing characteristics of the physical environment, such as voltages and temperatures, and in representing output signals by which they may be regulated. The notation used to represent a time-continuous flow is a line with a double arrowhead and a simple example is shown in Figure 2(a). Time discrete flows have values at individual points in time and have null values at other times and are represented by a line with a single arrowhead. Items in a data store remain invariant over periods of time between points at which processes change them. During these periods the contents are available to be used by other processes. Therefore a store represents a time-delayed relationship between processes. Since changes to and uses of a store occur at discrete points in time the data flows to and from a store are shown as discrete flows.

In general, systems also contain certain dataflows which have no real content but merely serve to modify the response of the system to incoming data. These are termed control flows and are normally represented by a dotted line. Control flows may be used to modify the state of a process, in the simplest case to turn it on or off (Figure 2(b)), or may be used to trigger a process (Figure 2(c)). The use of control flows therefore allows a rigorous interpretation of the behaviour of data transformations over time.

Figure 2 DFD notation used to indicate time dependency and control nature of data flows
(a) Time continuous flows are indicated with a double arrowhead
(b) Control flows (dotted lines) can change the state of the process they act upon
(c) Control flows can trigger the process they act upon
A transformation that accepts only control flows as inputs and generates only control flows as outputs is termed a Control Transformation. The control transformation must keep track of which control flows it has already received and use this information to help generate its outputs i.e. it must memorise the state of the system being controlled. At each point in time the internal memory and the input control flow determine the output control flow. The use of control transformations is particularly common in interactive applications where commands from the operator have to be interpreted and dispatched to activate the relevant procedure (Figure 3).

Figure 3 Data Flow Diagram of an interactive application illustrating use of control flows

One of the advantages of formulating the problem in terms of the DFD is that they can be analysed for mistakes. The following is a list of symptoms of a faulty analysis which may serve as a useful guideline for avoiding some of the more common mistakes made in drawing DFDs:

- Processes which create outputs from data they do not have
- Write-only files
- Complicated interfaces. Many dataflows between two processes probably indicate that the system has been partitioned badly
- Inaccurately and poorly named dataflows. This indicates that the true composition of the data is not known
- Poorly named processes. Processes should be named in terms of their input and output flows. Thus the dataflows should be labelled first and processes added afterwards. This is an effective way of producing a good decomposition because it shortcircuits our preconceptions as to what processes should be included
- Presence of implementation details since the aim is to generate a logical model which is valid for any choice of implementation
Obviously complex systems cannot be easily represented by a single DFD and some method of decomposing the problem into its various components is needed. Usually the first step in performing the decomposition is to delineate the boundaries of the system. This can be achieved by developing a context diagram showing all sources and sinks in the system’s environment and their interaction with the system. By convention the entire system activity is represented by a single transformation and flows internal to the system are not shown. An example of a context diagram for a hypothetical high energy physics experiment is shown in Figure 4.

![Context DFD for a hypothetical HEP experiment](image)

Figure 4  Context DFD for a hypothetical HEP experiment

Once this has been achieved the next step is to decompose the system into sub-systems until the primitive components are identified. There are several ways in which the decomposition may be carried out. Functional Decomposition, or the top-down approach, involves the stepwise decomposition of the problem into levels of greater and greater detail and results in a levelled set of DFDs (Figure 5). The decomposition is continued until the bottom level 'bubbles' (functional primitives) can be represented in about half a page of text. The amount of information presented at each level should be controlled to an amount which can be easily understood by the reviewers of the design. It has been shown\(^\text{13}\) that human error rate increases dramatically as the number of components of the problem which have to be dealt with concurrently increases. However if the problem can be broken into a number of components which can be dealt with separately then the error rate only increases linearly with the number of sub-problems (Figure 6). Therefore a rule of thumb guideline is that the number of processes in each DFD should lie within the range 7±2 (Miller’s Limits).

An alternative approach to decomposition which may be more appropriate to real-time applications is the Event List approach. In this case the list of events taking place in the system’s environment are specified and the ways in which the system must respond to those events are modelled separately. Note that events usually appear as control flows on the DFD, some examples being:

- Operator starts datataking
- Trigger received
- Event limit reached
- End of tape reached
- Detector high voltage out of range
Figure 5  Notation for System Decomposition into several levels of DFDs

Figure 6  Miller's Limits
Once the transformation schema has been derived from the event list then the event responses and associated data flows can be grouped into larger units to complete the upper levels of the design. Also the event responses can be partitioned into groups of lower level transformations to complete the lower levels of the design.

4.2 Specifying Transformations

Once the decomposition has been completed the processes appearing in the lowest level diagrams must be described precisely to state the way in which the flows entering the process are transformed into the flows leaving it. There are two different techniques used for dealing with the two types of transformation process.

The recommended way of describing data transformations is to use a restricted subset of a natural language, such as English, since full narrative text tends to be imprecise and ambiguous. The language, which is called the Process Description Language or PDL, should consist of imperative verbs, elements appearing on the DFDs and special keywords for formulating logic. It is very often targeted at the programming language which will be used once the design is to be implemented and is sometimes referred to as pseudocode.

The PDL for the process used to Build Events might look as follows:

```
Begin
  Get Trigger Mask
  Form Readout List
  Do while Readout List still has entries
    Read Digitising from next detector on list
    Remove detector from list
  End do
  Fill Event header
End
```

As already mentioned the control transformation must memorise the state of the system and each time an input appears on one of its input control flows must examine this memory to determine which output control flows to activate. The most suitable technique for specifying a control transformation is the State Transition Diagram (STD) which has been widely used in the past particularly for the design of network protocols. The components of the STD are:

- the state
- the transition between states
- the condition for the transition to take place
- the action taken as the transition occurs

The STD used to specify the control transformation shown in Figure 3 is illustrated in Figure 7(a). The states are represented on the diagram by labelled boxes and the permitted transitions between states by arrowed lines. Each line representing a transition is labelled with the condition for the transition to take place and the action to be performed in making the transition to the new state. By convention the condition and action are separated by a horizontal line to easily distinguish them. The information contained in the STD can also be represented in a tabular form (Figure 7(b)) which has the advantage that it forces the designer to consider the possibility of each condition occurring for every state of the system. This helps to avoid the problem of the system being unable to cope with unforeseen events when it is finally implemented and tested.
Figure 7 (a) State Transition Diagram For Specifying the Control Transformation "Invoke DAQ Request"  
(b) Corresponding State Transition Table
4.3 Modelling Data Structures

Certain composite data flows appearing on the DFDs can be extremely complicated and likewise certain data stores take on more the character of a database. Thus at the higher levels in the DFD hierarchy it is sometimes difficult to continue partitioning the system without having a much better idea of the content of the data being dealt with. These problems can be solved if a data model describing the individual data elements and their relationships is available.

The data model used in SASD uses the basic notation and concepts of the Entity - Relationship diagram\(^{14}\) which is just a graphical abstraction on the use of nouns and verbs in written sentences. The entities are the objects in the system and are represented on the diagram by rectangles with a title corresponding to the name of the entity. Each entity may have attributes and these are listed below the title inside the box. Relationships between entities are shown by an arrow going from one entity set to the other. The arrow may be labelled to indicate the precise form of the relationship or may be left unlabelled if the nature of the relationship is obvious.

A simple example including two entity sets, Fastbus Device and Fastbus Crate, is shown in Figure 8. The diagram may be interpreted as meaning "A Fastbus Device belongs in a Fastbus Crate". The simple arrow notation has been extended to reflect the cardinality of the relationship between these two entity sets. The double arrowhead indicates that there may be more than one Fastbus Device associated with a single Fastbus Crate. The bar on the left of the arrow indicate that a particular Fastbus Device may not belong in any Fastbus crate while the bar on the right indicates that there may be Fastbus crates which do not contain any Fastbus devices. This example also shows that relationships may have attributes. In this case Slot Number indicates in which slot of the Fastbus crate a particular device is located.

![Figure 8](image-url)  
**Figure 8**  An Entity - Relationship Diagram containing two Entity Sets, Fastbus Device and Fastbus Crate

It is natural to store all the data elements associated with an entity set in the form of a table. Each member of the entity set corresponds to a row in the table and each attribute of the entities is represented by a column. Each relationship may be represented by a column of one or the other entity sets it relates. This corresponds exactly to the form in which data are stored in a relational database and therefore advantage can be taken of the powerful features offered by modern commercial Relational Database Management systems (e.g. Oracle\(^{15}\)) for providing access to the data. In addition a group from Aleph has developed a system, called ADAMO\(^{16}\), to define Entity Relationship data structures, map them onto tables and manipulate them from FORTRAN programs. This system permits the efficient handling of numeric data within algorithms and has tools for transferring tables to and from the database managed by Oracle.

294
The data modelling technique is an invaluable aid to describing complex data structures. In Figure 9 a small part of the data model describing a Fastbus layout is shown. Such diagrams can be analysed to see if they offer a correct interpretation of the system being modelled (in this case the Fastbus standard). Since a data structure specification can be viewed as being correct or incorrect the program structure based on a data structure specification can often be viewed as being correct or incorrect. In some cases complete methodologies have been based on this principle which leads essentially to a bottom-up approach to system design.

![Diagram of Fastbus layout](image)

**Figure 9**  Extract from the Entity-Relationship diagram used to model Fastbus

### 4.4 Specifying data

All data flows and data stores appearing in DFDs are described in more detail in the Data Dictionary (DD) and as such the DD operates as the central reference point for all information about the system. An entry in the DD for a particular item would typically consist of a short description explaining the meaning of the item followed by an expression giving a more precise description in terms of its components or the values it is allowed to take. A simple mathematical-style notation is usually used to make the dictionary more readable. Some of the symbols commonly used are as follows:

295
is interpreted as meaning "is defined as"
+ indicates a sequence of data items used when describing composite data flows
[ ] indicates a selection of data items
{ } indicates iterations of data items
( ) indicates optional inclusion of a data item
* indicates the beginning or end of a comment

An extract from the DD containing definitions of some of the flows found in the DFD shown on Figure 1 may look as follows:

**EventRecord**

* Event formatted for storage on a physical device *

\[ \text{EventHeader} + \{\text{Event}\} + \text{EventTrailer} \]

**Event**

* Collection of digitisings from the complete experiment *

\[ \{\text{Digitisings}\} \]

**Digitisings**

* Digitised signals from one of the experiment's detectors*

\[
\begin{align*}
\text{TPCDigits} \\
\text{ECALDigits} \\
\text{HCALDigits} \\
\text{TOFDigits} \\
\text{MUONDigits}
\end{align*}
\]

**TPCDigits**

* Digitisings from the Time Projection Chamber formatted in a single bank containing both cathode pad and proportional wire information *

\[ \text{BankHeader} + \text{TPCPad} + \text{TPCWire} \]

Information from the data model may also be stored in the DD. As an example entries from the DD corresponding to some elements of the Fastbus data model are shown below. A simple notation is needed to describe entity sets, attributes of entities and relationships between entities. The example uses the ADAMO notation where firstly the entity sets are defined in terms of the list of their attributes enclosed in parentheses. Following this the attributes themselves are defined and any restrictions on the values they are allowed to take (in the form \([\text{min}, \text{max}]\)) may be specified. Finally the relationships are specified in terms of the names of the entity sets they relate and the cardinality of the relationship.
DEFINE ESET

Port
  * Geographically addressable part of a Fastbus device *
  = ( GA, DA, EnableLogicalAddressing)

DEFINE ATTRIBUTE

DA
  * Device Address *
  = INTEGER [0,*]

DEFINE RSET

  ( Port [1,1] -> [1,*] Segment )
  * A Fastbus port is identified by a geographical address on a specific segment *

Note that the relationship given in the above example may be interpreted as "one or more than one Port may be associated with a particular Segment (ie. [1,*]) whereas one and only one Segment may be associated with a particular Port (ie. [1,1])"

4.5 The Functional Specification

The result of the analysis phase is a complete validated specification of the required functions, interfaces and performance of the system. The deliverable items produced as a result of applying the Structured Analysis techniques afford two complementary views of the system, the Transformation Schema and the Data Schema. The Transformation Schema is represented by a levelled set of DFDs which together describe the principle data flows of the system and the processes that transform them into one another. The processes appearing in the lowest level diagrams (functional primitives) are described in terms of a set of mini-specifications, structured English being used for data transformations and State Transition Diagrams for control transformations. The Data Schema provides a model for all the data items in the system including the relationships between data items and is represented by a set of Entity Relationship diagrams. The two views are tied together by the Data Dictionary which contains more details and comments describing items appearing in the diagrams.

In addition to the technical methods described above there are also management procedures which can be used to establish the requirements for validation, verification and acceptance testing and also for producing a detailed project plan with a first outline of cost and schedule estimates. These management procedures are not described in these lectures but are well covered in the literature.
5) THE DESIGN PHASE

The aim of the design phase is to ensure a high quality for the software products that result. The two principles which are closely followed in Structured Design in order to achieve this goal are modularity and hierarchy. Modularity implies that the complete system should be decomposed into collections of program modules, each module defining a simple relationship between its inputs and outputs. Modules should be usable without the need to know how they work and they should always perform the same task. Hierarchy implies the use of hierarchical program constructs i.e. tree structures. Important decisions are made high up in the hierarchy whilst less important ones are taken lower down. It is very important that the tree structure should be balanced, i.e., that a single module should not control too many or too few subordinate modules, and that information should be passed between modules only on a need-to-know basis. By adopting this approach it should be possible to identify parts of the system which are both manageably small and solvable separately. In addition, the finished product should be easier to modify and maintain as the system evolves.

Thus the role of Structured Design is to translate the functional specification derived during the analysis phase into a software design incorporating a modular tree-structure of software procedures. This design phase is carried out in a number of well-defined steps. Firstly the architecture of the software modules must be defined and this is represented using another diagramming technique, the structure chart. This step involves translating the network of data flows and transformation processes present in the DFDs into a hierarchy of program modules. Following the overall design the internal logic of each module in the structure chart is then developed. Finally the detailed design considerations can be applied e.g. to modify the structure chart to take account of real-time constraints or to add modules to the structure chart needed for accessing data structures etc. The final product incorporating the structure chart, DD and module descriptions is the design specification.

5.1) Overall Design

The technique used for representing the architecture of the software system in terms of its software modules and their interfaces is the structure chart. Each software module is represented on the chart by a rectangular box labelled with the name of the module. Figure 10(a) illustrates the notation used to describe the procedure call interface between two modules. In this example module A calls module B and passes to it the two data parameters X and Y. On completion module B passes control back to module A together with the control parameter E. Procedural details can also be shown on the structure chart. For example in Figure 10(b) module A first calls module B and then module C and the curved arrow indicates that this sequence is to be repeated a number of times. In Figure 10(c) the diamond indicates that module A will either call Module B or module C according to the result of some decision it has to take.

![Figure 10 Notation used in the Structure Chart diagramming technique](image)
A completed structure chart always has a MAIN module at the top which is linked to the subordinate modules it controls positioned below it. These modules in turn call their own subordinate modules thus establishing the hierarchical tree-structure. On the structure chart sequencing is generally taken to be from left to right with the modules being organised broadly speaking into three branches, one for gathering information, one for processing, and one for outputting information.

Although in practice there is no automatic procedure for deriving the structure chart from the set of DFDs there are two different systematic approaches which can be used to define a strategy for making the transition. As will become apparent two approaches are needed to cover two basically different types of DFD topology.

In the majority of cases the first approach, so-called Transform Analysis, may be used. Transform Analysis consists of identifying the central transformation and of “picking up” the DFD by this transformation and letting the input and output flows hang down from the middle. Modules in the input and output streams may be factored into a GET module, a TRANSFORM module and a PUT module (Figure 11). Modules belonging to the input stream are termed afferent modules whereas those belonging to the output stream are termed efferent modules. DFDs suitable for processing by transform analysis are those where some dataflow throughput is detectable on the diagram, an example being the DFD shown in Figure 1.

![Diagram](image)

Figure 11  Deriving the Structure Chart using Transform Analysis
In those cases where it is not possible to identify the central transformation this technique cannot be used. This situation occurs when the input stream is split into a number of substreams each of which is treated differently (Figure 12). In this situation the second approach so-called Transaction Analysis can be used. The process responsible for splitting the input stream is termed the transaction centre. Once a transaction centre has been identified on a DFD the scheme shown in Figure 12 should be used for organising the corresponding structure chart. This scheme contains a module for obtaining the coded transaction and a second module for dispatching the transaction to the relevant module for processing. In those cases where a DFD shows both characteristics then the two analysis strategies can of course be combined.

Figure 12 Deriving the Structure Chart using Transaction Analysis
Two criteria called cohesion and coupling have been developed for evaluating the quality of the design represented by the structure chart. Cohesion measures the strength of association of the different elements within a particular module and therefore a high degree of cohesion is desirable in a good design. The following qualitative measures may be assigned to indicate the degree of cohesion, the first being the worst form of cohesion leading to the last which is the best:

- Coincidental - there is no relationship between the component parts of a module
- Logical - a module performing several functions all similar in some respect, e.g. do all input
- Temporal - a module whose functions are all performed in a short space of time e.g. initialisation
- Procedural - describes modules which are inextricably linked and cannot be used independently
- Communicational - results when functions that operate on common data are linked together
- Sequential - describes a module whose functions are to make a number of transformations which must be performed in sequence because of data dependency
- Functional - every component within a module contributes to perform one single function e.g. SQRT

Coupling measures the interdependence of modules and the extent to which they are connected. The aim is to produce loosely coupled systems since in this case modifications to one module have little or no influence on modules connected to it. Thus loosely coupled systems are much easier to maintain as changes are made during the evolution phase of the system. An obvious measure of the degree of coupling between modules is the complexity of the interface between them. In loosely coupled systems very few parameters are exchanged between modules. Another factor which can strongly affect coupling between modules is the presence of a common data area. If the operations performed by a particular module on the global data area are changed then the other modules linked to that area may cease to function correctly. In addition if the structure or meaning of the data in the global area itself are changed then all modules accessing the data area will be affected.

The problems associated with global data areas can be minimised either by avoiding their use altogether and passing information explicitly through parameter lists or by restricting access to the global data area to a special set of access procedures. This last approach is termed Information Hiding and the set of access procedures are together referred to as the Information Hiding Module (Figure 13). The design of the Information Hiding Module...

![Diagram](image)

Figure 13 Illustration of the packaging of data access routines in an Information Hiding Module
Module may be separated from the rest of the system and treated in semi-isolation by one of the members of the design team. This is a good example of a situation in which a data model describing the contents of the data area should be produced before starting the design of the system. The data model together with a list of valid operations to be performed on the data will very often provide sufficient information for the design of the access procedures to be made.

Clearly there is a correlation between the two evaluation criteria in that modules exhibiting a high degree of cohesion will also tend to be loosely coupled. The example shown in Figure 14 may help to illustrate by analogy this correlation.

**Figure 14** Illustration of relationship between Cohesion and Coupling by analogy with the distribution of establishments within an imaginary town

### 5.2 Module Design

The same techniques are used for specifying modules as were mentioned in Section 4.2 for specifying transformations. In many cases modules appearing in the structure chart can be easily identified with processes appearing in the DFDs and so only descriptions for new modules need to be provided. As mentioned previously the descriptions can be targeted towards the programming language being used or if desired can be implemented in code directly.
5.3 Design Optimisation

The system represented by the structure chart must run in a real environment and so several different constraints may need to be satisfied. It is important that these constraints should not unnecessarily compromise the design and so they are applied only once the initial design has been completed. The fact that the fundamental design principles have been followed should make it easier to rearrange the modules belonging to the structure chart to satisfy realistic requirements.

A typical example of a real-time constraint is the limit put on the time taken for the system to respond to a predefined stimulus. Figure 15 shows an example of how modules in the structure chart can be repackaged to provide a response in an adequate time in this case by eliminating the need for some of the middle level modules. Although in some cases more drastic action may be required this method of satisfying the constraint is to be preferred to changing the detailed structure of individual modules.

![Diagram of a structure chart]

Figure 15 Repackaging the Structure Chart to provide a real-time response

In addition the temporal relationships between different parts of the program structure can be analysed to look for preferred groupings of modules. In Figure 16 for example the two major functions of the system, namely to build and plot histograms, have been separated by the introduction of a few extra program modules. The modified design can now be more easily implemented to permit these two functions of the system to be activated asynchronously.
Figure 16  Producing preferred groupings of modules by analysis of their temporal relationships

6) THE IMPLEMENTATION AND TESTING PHASES

Once the design has been completed, there are several ways in which it can be implemented and tested. The first choice to be made is whether to adopt a phased or an incremental approach. In both cases, the first step is the coding and testing of each program module in isolation. Once this step has been completed, the modules need to be combined and the combinations tested, and this is where the two approaches differ. In the phased approach, large groups of modules, such as the complete system or a subsystem or a complete level in the structure chart, are combined and tested at once. In the incremental approach, firstly two modules are combined and tested, and subsequently new modules are added progressively one at a time to the combination until the whole system is complete and working. The incremental approach clearly offers a more controlled method for tracing errors and for making thorough tests.
A second choice to be made is at which level in the structure chart to begin the implementation of the system. In the bottom-up approach the lowest level modules are implemented first and are tested using "test drivers". Subsequently the test drivers can be replaced by the correct coordinating modules until the subsystem is complete. Finally the subsystems are combined. In the top-down approach the highest level modules are implemented first and are tested using "stubs". Gradually the stubs are replaced by the correct subordinate modules until the system is complete. The principle benefits of taking the top-down approach are that system testing and integration are done throughout the project which means that users can see and use working prototypes. However the bottom-up approach may be more appropriate in real-time applications for the testing of critical hardware where it is often difficult to generate test data. In addition the bottom-up approach has the advantage that, for systems being implemented by a team, different programmers can be allocated immediately to different parts of the system.

Care must be taken in the choice of test procedures to ensure that the system is subjected to a comprehensive set of checks. In addition to logic errors the program must be tested in different conditions to look for errors caused by random events in the system's environment and to check recovery procedures in the case of hardware failures or the presence of invalid input data. Since testing involves searching for the presence of errors it can never be really completed. However a minimum requirement should be that all modules in the system have been executed at least once even though it may not be possible to test all possible execution paths through the program.

It is important to note that many automated tools have been written for enhancing a programmer's productivity in the implementation and testing of software and some of these tools are described in more detail in the next section.

7) AUTOMATED TOOLS

Automated tools are now being used quite routinely to reduce labour-intensive activities associated with software production and thereby to improve the productivity of programmers and the quality of the software products developed. The range of automated tools available can be used to capture the design specification in a central database and to trace its development through all stages of the software life-cycle. At any time documentation reflecting the current status of the project can be generated directly from the database which provides an extremely useful way of exchanging information between members of a design team as well as for monitoring progress made in the project. The tools available will be considered in three different categories:

7.1) Manager's Toolbox

Although procedures for managing projects have not been treated in these lectures it is worth noting that many tools are available to help in the management of project resources and project expenditure. Examples include the electronic spreadsheet as well as tools for maintaining directories of information, such as network addresses. In this latter case facilities offered by database management systems can be extremely useful for developing customised tools.

7.2) Designer's Toolbox

There are a number of commercial products available for supporting the technical methods used by SASD for the analysis and design of software\textsuperscript{17}. However the development of these tools by commercial software companies has only accelerated rather recently and so several organisations who started to use these techniques some time ago have developed their own packages (e.g. Boeing and Lear Siegler). This has also been the case in the Aleph experiment where a comprehensive set of tools has been developed to support the collaboration's use of SASD techniques\textsuperscript{16}. 305
The basic facilities offered by these packages include graphics editors for entering the different types of diagrams and language sensitive editors for entering the data specification in the Data Dictionary. They also provide for the automatic checking of consistency between connected diagrams as well as between diagrams and the Data Dictionary. In some cases they attempt the automatic generation of the structure chart from the set of Data Flow Diagrams. Note that although PCs are ideal for entering and manipulating software designs it is often important that the captured design should be centrally available so that it may be accessed by colleagues who in some cases may be situated remotely and therefore require access through wide area networks.

7.3) Programmer’s Toolbox

There has been considerable investment by computer manufacturers and software companies in providing a great variety of tools to support the coding, testing and optimisation of software. In this section a number of tools are discussed which in the author’s experience have proved to be particularly productive and convenient to use. It will be noticed that some emphasis is placed on describing those available as part of DEC’s Software Engineering Tools package, VAXset.

The use of syntax directed editors for entering source code is now quite common. These editors understand the syntax of supported languages and provide templates for the standard constructs of the respective languages. In addition these templates can be customised to recognise the syntax of new applications, such as the DD syntax, and to establish a uniform programming style for program layout, program documentation and coding conventions. The convenience and efficiency with which code can be developed can also be improved eg. in the case of DEC’s new editor (LSE) several source files can be edited in parallel in separate “windows” on the terminal screen and also the compiler can be invoked directly from the editing session.

It is often advisable to restrict use of a programming language to a certain set of conventions for a number of reasons:

- to ensure portability, which is needed in the case that it is required to run the software on more than one type of computer system; in this case it is important to adhere to the standard specification of the language and avoid using extensions to the language provided by the machine on which the code is developed.
- to avoid misuse of the language and thereby reduce the likelihood of introducing errors.
- to encourage a healthy programming style which enhances readability and makes maintenance of the finished product that much easier.

For example, at CERN the FLOP program can be used to check FORTRAN programs against a mandatory set of rules. The program can also be used to improve the visual appearance of the source, eg. by indenting relevant pieces of the code and by resequencing statement labels etc.

Source code management systems are used to keep track of the history of changes to a piece of software. They permit the maintenance of multiple versions and keep track of what changes were made to the source, who made the changes and when they were made. The code management system PATCHY was developed at CERN and has been used successfully by a large number of experiments since it was introduced approximately ten years ago. More recently commercial products have come onto the market, such as HISTORIAN PLUS (from Opcode) and CMS (from DEC), and these are now in widespread use in the HEP community.

In large systems where the component program modules can be distributed among many separate files keeping track of file dependencies can be a major problem. The UNIX MAKE facility and DEC’s Module Management System (MMS) will regenerate a module only if one of its dependent modules has been modified since it was last built. The list of file dependencies is maintained by the programmer and supplied to the utility as a special input file. In this way relationships between load modules, object modules and source modules can be defined and used to rebuild the complete system in such a way that only those modules which need to be regenerated are rebuilt.
The most basic way of testing software products is by means of the debugger provided by the manufacturer of the machine being used. Under VAX/VMS the debugger is actually linked to the program under test and control of the execution of the program is given by the debugger to the operator. The operator then has the opportunity to step through the program and after each instruction has been executed examine the contents of various quantities eg. symbols from the program, registers etc. The current position in the source of the program is displayed on one part of the terminal screen whilst output from the debugger is routed to another part of the screen. In addition the program can be told to execute until a particular break point is reached when control is passed back to the operator. The debugger was made to be compatible with DEC's language sensitive editor in that the operator can step directly from the debugger to the editor to modify any fault in the source discovered during the test session. DEC also provide a tool for regression testing which attempts to ensure that new features do not affect the correct execution of previously tested features. This involves storing the results of each system test and comparing them with those obtained with the previous test. If differences occur then the software being tested is said to have "regressed".

Finally one of the most important features of the finished product is its performance. The performance of the software can be measured in a number of ways eg:

- by determining how much time is spent in each region of the program
- by looking at input/output activity
- by looking at the page fault rate

The results of such a performance analysis can be used to identify which region of the program should receive special attention when optimising the code in order to achieve the biggest improvement in overall performance. DEC's Performance and Coverage Analyzer (PCA) measures performance by sampling the program counter every 10 msec and storing the results, together with the other analysis information, on a disk file. These results may subsequently be read from the file and presented on the screen in histogram form. PCA may also be used to monitor which code paths are being exercised when tests are executed.

7.4) Future Prospects for Automated Tools

The ultimate goal in the development of these packages is the ability to generate code automatically from well structured designs (so called "end-user programming"). Several companies are experimenting with the use of expert systems to aid in the development of software systems \(^{18}\). The software development cycle now starts with the user entering a specification. This specification is analysed according to rules stored in a knowledge database following which it is decomposed and refined into a more detailed structure. These last two steps are iterated until finally the specification emerges written in the target programming language. Industry is taking such developments very seriously although it is likely that some considerable time will elapse before these products come into widespread use.

8) CONCLUDING REMARKS

There is plenty of evidence to support the view that structured methods have been employed extremely successfully in industry. Adopting the approach outlined in these lectures results in far more time being spent in the analysis and design phases of the project life-cycle and experience has shown that this leads to better software products. Documentation in the form of diagrams and mini-specifications is generated automatically and can be used as a powerful means to abstract and show details. As such it can be used as a basis to review the status of the project throughout the development process. However the High Energy Physics community has only recently started to get to grips with these techniques and it is fairly clear that a certain discipline has to be demonstrated by members of the software development team if this approach is to be successful.
9) Acknowledgements

I owe a special debt of thanks to many of my Aleph colleagues for many illuminating discussions over the last year or so during which time we endeavoured to become "disciples" of the SASD methodology. In particular I must thank Gottfried Kellner for supplying me with much background material which proved to be invaluable when preparing these lectures and to Paolo Palazzi and Steve Fisher from the ADAMO team for passing on some of their expertise on data modeling techniques. The Fastbus data model was designed by Richard McClatchey and I would like to thank him for permission to reproduce a simplified version of his diagrams in these proceedings. Much of the material used to prepare these lectures came from a course given by Alan Kennedy attended by myself and many other people at CERN. I am sure that the quality of his lecturing had a great influence on the decisions taken by collaborations to try using these techniques in their software projects.

References
2) F.P.Brooks Jr., The mythical man-month, Datamation, Dec 1974
3) E.W.Dijkstra, GO TO considered harmful, Communications of the ACM, Vol 11, No 3 (May 1966), 147-148
6) D.L.Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Vol 15, No 12, (Dec 1972) 1053-1058
13) G.A.Miller, "The magical number. Seven plus or minus two. Some limits on our capacity for processing information", Psychological Review, Vol 63, No 2, (March 1956)
14) P.P.Chen, "The Entity - Relationship Model - Toward a unified view of data", ACM Transactions on Database Systems, Vol 1, No 1, (March 1976), 9-36
15) The ORACLE Relational Database Management System, Oracle Corp., Belmont, Calif., USA
16) The various components of the ADAMO system are described in a number of ADAMO notes which can be obtained from P.Palazzi, DD Division, CERN, CH-1211 Geneva 23
17) Some of the tools developed for support of SASD techniques are: ProMod available from GEI, D-5100 Aachen, W.Germany
SATOOLS and SDTOOLS available from Tektronix, Beaverton, Oregon 97075
Argus II developed by Boeing Computer Services for internal use, Seattle WA 98124
StructSoft a PC based tool for SA, 24 Homer ST., Parsippany, NJ 07054
Teamwork/SA from Cadre Technologies Inc., 222 Richmond St., Providence, RI 02903
18) A.Wolfe, Bringing order out of chaos in software design, Electronics, Nov 4, 1985

308
LIST OF PARTICIPANTS

SPEAKERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRUINS, T.</td>
<td>Rijkswaterstaat, 's Gravenhage</td>
<td>Netherlands</td>
</tr>
<tr>
<td>CARPENTER, B.</td>
<td>CERN, Geneva</td>
<td>Switzerland</td>
</tr>
<tr>
<td>CASIMIR, H.B.G.</td>
<td>Eindhoven, Netherlands</td>
<td></td>
</tr>
<tr>
<td>CITTOLIN, S.</td>
<td>CERN, Geneva</td>
<td>Switzerland</td>
</tr>
<tr>
<td>DE SWAAN ARONS, H.</td>
<td>Delft University of Technology</td>
<td>Netherlands</td>
</tr>
<tr>
<td>HARVEY, J.</td>
<td>Rutherford Appleton Laboratory, Chilton, U.K.</td>
<td></td>
</tr>
<tr>
<td>HERTZBERGER, L.O.</td>
<td>NIKHEF-H, Amsterdam</td>
<td>Netherlands</td>
</tr>
<tr>
<td>HOOGLAND, W.</td>
<td>NIKHEF-H, Amsterdam</td>
<td>Netherlands</td>
</tr>
<tr>
<td>KATEVENIS, M.</td>
<td>Research Center of Crete, Heraklion, Greece</td>
<td></td>
</tr>
<tr>
<td>KLINT, P.</td>
<td>CWI, Amsterdam</td>
<td>Netherlands</td>
</tr>
<tr>
<td>MOUNT, R.P.</td>
<td>Caltec, Pasadena, U.S.A.</td>
<td></td>
</tr>
<tr>
<td>NIERSTRAS, O.</td>
<td>Geneva University, Switzerland</td>
<td>Switzerland</td>
</tr>
<tr>
<td>PELLEGRINI, C.</td>
<td>Geneva University, Switzerland</td>
<td>Switzerland</td>
</tr>
<tr>
<td>PYLE, I.C.</td>
<td>Systems Designers Scientific, Farnborough, U.K.</td>
<td></td>
</tr>
<tr>
<td>SEIS, I.</td>
<td>CERN, Geneva</td>
<td>Switzerland</td>
</tr>
<tr>
<td>TINDEMANS, P.</td>
<td>Ministry of Education &amp; Sciences, 's Gravenhage, Netherlands</td>
<td></td>
</tr>
<tr>
<td>VAN DAM, A.</td>
<td>Brown University, Providence, U.S.A.</td>
<td></td>
</tr>
<tr>
<td>VASCOTTI, A.</td>
<td>CERN, Geneva</td>
<td>Switzerland</td>
</tr>
<tr>
<td>VERHAART, G.J.</td>
<td>OSI-Holland, Eindhoven</td>
<td>Netherlands</td>
</tr>
<tr>
<td>VERMAASEREN, J.</td>
<td>NIKHEF-H, Amsterdam</td>
<td>Netherlands</td>
</tr>
<tr>
<td>VERMEULEN, J.</td>
<td>NIKHEF-H, Amsterdam</td>
<td>Netherlands</td>
</tr>
<tr>
<td>VON RÜDEN, W.</td>
<td>CERN, Geneva</td>
<td>Switzerland</td>
</tr>
</tbody>
</table>
STUDENTS

ANDERSSON, A.-L. CERN, Geneva, Switzerland
ASKEW, C. Southampton University, U.K.
BARRAND, G. LAL, Orsay, France
BEE, C. Oliver Lodge Laboratory, Liverpool, U.K.
BILLAT, C. Ecole Polytechnique, Palaiseau, France
BRAIBANT, S. Université à l’Etat de Mons, Belgium
BUNN, J. CERN, Geneva, Switzerland
CASALEGNO, L. CERN, Geneva, Switzerland
CHIU, P.C.M. Rutherford Appleton Laboratory, Chilton, U.K.
DEFFUR, E. RWTH Aachen, Fed. Rep. of Germany
DE JODE, M. I.I.H.E., Brussels, Belgium
DELIKARIS, D. Collège de France, Paris, France
DONG, F. University of London Computer Centre, U.K.
DUFOUR, M. CERN, Strasbourg, France
ESTEVENY, L. CERN, Geneva, Switzerland
FUTO, E. CERN, Geneva, Switzerland
GILLEY, T. Weizmann Institute of Science, Rehovoth, Israel
GOMES, P. CFCM/INIC, Lisbon, Portugal
GRANDJEAN, Y. CERN, Geneva, Switzerland
HAYMAN, J. SIN, Villigen, Switzerland
JOHANSSON, S. Lund University, Sweden
JOSEPHUS JITTA, D. NKHEF-H, Amsterdam
KAPITZA, H. Carleton University, Ottawa, Canada
KING, B. Liverpool University, U.K.
LEVY-MANDEL, A. CERN, Geneva, Switzerland
MASONI, A. Cagliari University, Italy
MCMILLAN, J. Leeds University, U.K.
MOGHILNY, G. L.P.N.E., Paris, France
MONTANANA, R. Instituto de Fisica Corpuscular, Valencia, Spain
MORANDIN, M. Padua University, Italy
MORPURGO, G. CERN, Geneva, Switzerland
NIEUWENHUIS, C.H.M. CERN, Geneva, Switzerland
ORTU, S. Universitat Autonoma de Barcelona, Spain
PACE, E. Laboratori Nazionali di Frascati, Italy
PELLEGRINO, M. CERN, Geneva, Switzerland
PETRILLI, A. CERN, Geneva, Switzerland
PIJLROMS, B. Edinburgh University, U.K.
PUN, T. CERN, Geneva, Switzerland
RASMUSSEN, J.T. Institute of Datalogy, Copenhagen, Denmark
RUHM, W. CERN, Geneva, Switzerland
SPANG, K. Niels Bohr Institute, Copenhagen, Denmark
VAN 'T VEEN, R. NKHEF-H, Amsterdam, Netherlands
VAN UITERT, B. DESY, Hamburg, Fed. Rep. of Germany
WEIHS, W. Siegen University, Fed. Rep. of Germany
WICHERS, V. V.V.I., Groningen, Netherlands
ZACHAROV, I. NKHEF-H, Amsterdam, Netherlands

310
PARALLEL PROCESSING USING TRANSPUTERS OR OCCAM (OR BOTH)

C.R. Askew

Department of Physics, University of Southampton, United Kingdom

ABSTRACT
This paper surveys the field of multi-transputer supercomputers, both presently available and planned. The programming language occam is described, and other languages which will soon be available for the transputer are mentioned. Programming strategies for large numbers of processors are presented.

1. INTRODUCTION

Scientists are always looking for more speed from their computers. The next generation of supercomputers will be of two different forms. The new vector machines will be faster than the last generation. They will still be expensive to both buy and maintain. The other type of supercomputer will contain many processors. Individually these processors will not be amazingly quick, but they will be able to work together on the same problem. A large collection of these processors will thus be capable of 'supercomputer' performance. The cost of producing such a computer can be greatly reduced if the processing unit is simple enough to be mass produced.

2. THE TRANSPUTER: AN INTRODUCTION

The T414 Transputer is a 32 bit microprocessor made by INMOS. There is however more than just a processor on the chip. The T414 also includes 2 kilobytes of fast static RAM, a programmable interface for external memory, and four INMOS links for communicating to other transputers.

The processor is of the reduced instruction set type. Instructions consist of 4 bits function and four bits data. This format implements the 15 most used instructions in a single byte. The prefix instruction extends the data field of any instruction by 4 bits. Using the prefix and the operate instructions, all other instructions not requiring data can be implemented in two bytes.
Examples of single byte instructions: load/store local, load constant, jump/conditional jump
add/subtract, prefix/negative prefix ...

There are only six registers: A, B and C form an evaluation stack, I is the instruction pointer, O is the operand register, and W is the workspace pointer. This simplicity allows context switching to be very fast, and so gives good support for concurrent program execution.

The fastest currently available transputers have a clock period of 50 nanoseconds. This gives a peak instruction rate of 10 MIPS (Millions of Instructions Per Second) for simple instruction sequences. This translates to a floating point performance of 0.09 megaflops (millions of floating point operations per second).
For floating point performance there is a new transputer shortly to be launched (sometime in 1987). This will have a floating point co-processor on the chip, along with a few other improvements. The floating point performance (according to INMOS) will be 1.0 megaflops.

The INMOS links can work at 5, 10 or 20 million bits per second. They use a serial protocol and only require two wires to be connected between the processors which are communicating. This implements two way communication between these processors. For longer distance communications (up to 0.5 meters) twisted pairs are used. Link transfers are set up by the processor but do not require processor activity during the transfer. The serial links provide a fast way of communicating to other processors and so give direct support for programs which run on more than one processor.

One very big advantage of the transputer is that it needs very little support logic. So little that 42 transputers (without external memory) can be put on a double extended eurocard.

The T414 is described in more detail in the 'Transputer reference manual' [1].

3. MULTI-TRANSPUTER MACHINES

3.1 Presently available

FPS T-series

The FPS T-series machines consist of transputers connected in a hypercube. Each T414 has a Weitek floating point vector processor capable of 8 megaflops add/subtract and 8 megaflops multiply, giving a peak performance of 16 megaflops. Special procedures are provided for access to the vector operations. A 64 node machine with a peak performance of 1 gigaflop has a price tag of approx £800,000. So this is a high performance machine with a low price tag iff the programs to be run on it can use the vector co-processors properly.

MEIKO "Computing Surface"

These computers consist of a rack (different sizes available) with a bus backplane, into which different boards can be plugged. A "Host" board consists of a T414 with 2 megabytes of dynamic RAM, with communications to the Host computer via IEEE 488 or RS232. This board also controls a bus on which all the other transputers lie, which allows communication (at a slower rate) between transputers not connected by their links. A "Compute" board consists of four T414 transputers each with 256 kilobytes of Dynamic RAM. A "Graphics" board has a T414 with some dual port video RAM, and the associated electronics to convert this bit map to RGB output. The cost of this machine is approx. £30,000 for four compute boards and one host board.

INMOS "ITEM 400"

The INMOS Transputer Evaluation Module consists of ten 'B003' boards. A B003 is a double extended eurocard with four T414 transputers, each with 256 kilobytes RAM, connected in a square with eight links coming out:

```
  0---1---2---3
```

312
This may be a disadvantage because some network topologies can not be achieved with this fixed arrangement of links. Also included in the price of £40,000 is a transputer development system for the IBM PC (or a lookalike), consisting of a 8004 board (T414 + 2 megabytes RAM) which slots into the back of the PC, and the software for program development, downloading and debugging.

3.2 Future products

INMOS M212
This is a 16 bit transputer specifically designed to control Winchester and floppy disks. In addition to the processor it has two links, 4 kilobytes of ROM and 1 kilobyte of RAM, an interface to external RAM and two disk interfaces. Once again a minimum of external logic will be required to build a disk system using a M212.

INMOS T800
This is the floating point enhanced transputer. It will have a floating point co-processor capable of more than a megaflop built-in. Also the T800 will have 4 kilobytes of internal RAM, an improved link interface, and some changes to the microcode to improve the graphics performance. It will be fully pin compatible with the T414 and will just need a new compiler which will utilise the co-processor for floating point arithmetic calls.

INMOS 32x32 link switch
Very little information about this link switch has been released. What is known is that it will switch 32 input links to 32 output links and that the link configuration will be set by a control link. There will be a small delay in propagating the link signal through the switch. No time scale has been announced for this possible future product.

MEiko switch
This switch, when it arrives will plug into the boards of a MEiko box, allowing electronic configuration of the links. This will make programming, or rather running programs much easier as it is fiddly and time consuming to wire up and check all the links for programs using different link configurations.

MEiko fortran farm board
Another addition to the range of boards is a single T414 with 8 megabytes of dynamic RAM on one board. This is designed with the execution of Fortran programs in mind. The idea is that several such boards can be running different programs, (or the same program with different data) and that they should share resources such as disks.

Supermodes: an ESPRIT project
Esprit project PI085 "A high performance low cost supercomputer" aims to utilise the power of the transputer. There are two main aims, a powerful single node workstation, and a multi-node machine with a performance in the gigaflop range but a price tag of less than £1 million. The project involves the departments of Physics and Electronics & Computer Science at Southampton University, INMOS, Royal Signals and Radar Establishment, Thorn EMI, Telmat, Apsis and Grenoble University. It covers the following areas: development of the T800, development and manufacture of the
supernode machine, high-level languages, debugging, and applications in the areas of physics, signal processing, graphics, real time programming, computer aided design, engineering, and manufacture.

4. LANGUAGES FOR THE TRANSPUTER

At present the only language available for multi-transputer machines is occam [2]. Fortran-66 BCPL and various assemblers exist for single transputers. In the near future, C, Pascal and Fortran-77 compilers will be released by INMOS. These compilers will enable compilation of units to be run from within occam, thus allowing multi-transputer execution. Channel I/O from within each 'alien' language is provided by means of predefined procedures. This means that C, Pascal, Fortran and occam modules can be run together, communicating by channels, either on the same transputer or on many transputers. Occam 2 [3] is also due to be released 'soon', the biggest improvements being in the handling of real numbers and arrays.

5. OCCAM : AN INTRODUCTION

5.1 General

Occam consists of 'Communicating Sequential Processes', and has close links to the language of the same name (CSP) [4]. Processes themselves are sequential, but can be run in parallel with other processes. Communication between these processes running in parallel is effected by point to point 'channels'. Sharing variables between concurrent processes is only allowed if none of the processes can write to that variable. This means that the only possible method of communication is a channel. Channels can be mapped to be internal (between processes on one transputer) or external (between processes running on different transputers, via a link). This means that the transition from multi-process execution on one processor to multi-processor execution of the same basic program is much easier.

Recursion is not possible in occam at the moment, but this is just due to the compilers available, there is nothing in the language which makes it impossible to implement a compiler which could deal with this.

Occam is such a simple language that there are a number of mathematical program transformations possible [5]. Algorithms which have been developed from mathematical definitions in this way are proven correct. The software for floating point operations was produced and verified in this manner.

5.2 Typing

The specific examples of occam are all in occam 2 rather than occam 1 for readability reasons. The main difference between the two versions of the language is that occam 2 has strong typing, and always requires explicit type conversions. The types are:
### 5.3 Conventional Constructs: SEQ, WHILE, IF, PROC

**SEQ**: do the following statements one after the other: -

```
SEQ
  i := i + j  -- note that the statements to which this
  j := j * 2  -- command refers are indented by two spaces
```

**WHILE**: a Pascal-like while loop: -

```
WHILE i < 10
  SEQ
    j := j + i
    i := i + 1
```

**IF**: a combination of both IF and CASE statements from Pascal. Really just a series of boolean expressions, each of which has some code associated with it. The first boolean to be true has its code executed. The program will cease to execute if none of the boolean expressions are true, thus the last boolean is normally 'true' to avoid this ever occurring.

```
IF
  i < j
  SEQ
    -- do something  ( -- is the comment symbol )
  i = j
  SEQ
    -- do something else
  TRUE  -- ( i > j )
  SEQ
    -- do something different
```

**PROC**: Procedure definition: -

```
PROC ToThePower ( REAL32 X, VAL INT Power )  -- VAL is not modifiable
  REAL32 Xin:
    -- local variable
  SEQ
    Xin := X
    SEQ LoopCount = 0 FOR Power
        -- replicated SEQ
        X := X * Xin
```

---

*a colon signifies the end of a definition*
5.4 **New Constructs** : CHAN, PAR, ALT, PRI PAR, PRI ALT, PLACED PAR, STOP

CHAN : defines a channel, may have a type associated with it :-

PROC Program (CHAN OF INT Keyboard, Screen)
  -- Keyboard and Screen are attached to the outside
  -- world and can only have integers sent along them

  INT Key:
  SEQ
  Screen ! (INT '?)
  WHILE TRUE
  SEQ
  Keyboard ? Key
  Screen ! Key
  : -- infinite loop

  -- get a key hit
  -- echo it to the user

  : -- end of procedure definition

PAR : do the following processes at the same time (time sliced) :-

CHAN OF BYTE ss: -- channel communicating between the processes
BYTE Size: -- Byte variable
PAR
  ss ! "*1Hello" -- first process (output bytes in string down ss)
  -- *1 is a byte depicting the length of the string
  SEQ
  ss ? Size
  SEQ Index = 0 FOR (INT Size)
  BYTE Ch:
  SEQ
  ss ? Ch
  Screen ! (INT Ch)

  -- second process
  -- read first byte (*1 - size)
  -- for loop
  -- local

  -- input next byte in string
  -- output as an INTeger to Screen

ALT : alternative, takes input from the first channel to be ready :-

VAL INT Max IS 100,
  Min IS 0:
INT Volume, ANY:
BOOL Continue:
SEQ
  Continue := TRUE
  Volume := Min
  WHILE Continue
  ALT
  (Volume > Min) & Quiet ? ANY
  Volume := Volume - 1
  (Volume < Max) & Loud ? ANY
  Volume := Volume + 1
  Stop ? ANY
  Continue := FALSE

  -- inputs on Quiet channel
  -- decrease the volume
  -- inputs on Loud chan
  -- increase the volume
  -- an input on Stop chan

  -- terminates the process

PRI ALT : prioritised alternative
In an ALT statement, when several of the channels become ready at the same time, it is undefined as to which one will be chosen. This is a problem if one channel is

316
always ready to input whenever the ALT is executed it may always be the channel to be chosen, irrespective of the state of the other channels. The PRI ALT statement always looks at the channels in textual sequence, so if two were to become ready at once, the higher one will be chosen. It is identical to the ALT statement in all other respects.

PRI PAR : prioritised parallel

The action of the PRI PAR statement is implementation dependant. For the transputer, there can only be two processes, the first one is at high priority and the other at low priority. In essence if a high priority process becomes ready to execute whilst a low priority one is executing, then the high priority process will interrupt the low priority one.

PLACED PAR : places seperately compiled processes onto processors :
-- definition of 'UserInterface' as a seperately compiled procedure
-- definition of 'DoWork' as a seperately compiled procedure
VAL INT LinkOut IS 0,
LinkIn IS 4:
CHAN UserOut, UserIn: -- channels running along link
PLACED PAR
PROCESSOR 0 T4 -- T4 is the processor type
PLACE UserOut AT LinkOut:
PLACE UserIn AT LinkIn:
UserInterface( UserOut, UserIn )

PROCESSOR 1 T4
PLACE UserIn AT LinkOut:
PLACE UserOut AT LinkIn:
DoWork( UserIn, UserOut )

Pictorially:

```
          "UserInterface"
          |___________|
          |        |
          |        |
          |        |
          |        |
          |        |
          |        |
          |___________|
          |        |
          |        |
          |        |
          |        |
          |        |
          |___________|

          "DoWork"
```

STOP : a process which starts but never stops!

This is useful for bringing things to a grinding halt. Executing a STOP simply ensures that this particular process will not proceed again, and that it will not take up any more processor time.
5.5 Von-Neumann vs Dataflow vs Demandflow

Occam is not restricted to just one of these, in fact it can make use of all three. Take the following example: a number of integers are to arrive down one channel, they are to be transformed and output down another channel.

The Von-Neumann approach is to determine how many numbers there are going to be, and to use this as a counter in a loop:

```
SEQ i=0 FOR NoOfNumbers
    SEQ
        In ? X
        Out ! ((X * X) + 1)
```

The Dataflow approach would be to precede every data value by a boolean, true for this being valid data, and false for termination:

```
BOOL Cont:
    SEQ
        In ? Cont
        WHILE Cont
            SEQ
                In ? X
                Out ! TRUE; ((X * X) + 1)
                In ? Cont
                Out ! FALSE
```

The Demandflow approach is to have a request channel for every data channel. If the request is true then send some data, if it is false, then send false back up the chain and terminate:

```
BOOL Cont:
    SEQ
        DemandingOutput ? Cont -- wait for demand/terminate
        DemandingInput ! Cont -- send on demand/terminate
        WHILE Cont -- while not terminated
            SEQ
                In ? X -- do transaction input
                Out ! ((X * X) + 1) -- and output
                DemandingOutput ? Cont -- demand/term as before
                DemandingInput ! Cont -- pass on
```

6. A FOLDING EDITOR: "every programmer should have one"

The document being edited can have folds inserted into it. A fold hides all lines within it, and just displays a comment line as a reminder as to what is inside. There are two ways of displaying the contents of such a fold, firstly if the fold is 'Opened' then the folds contents are put onto the screen below the fold comment line, and all the text which was below the comment line before is moved down the screen.

```
| line 1
| line 2
|    ... fold line
| line 3
```

when opened looks like:

```
| line 1
| line 2
|{{ fold line
| text within fold 1
| text within fold 2
|    ... fold within fold
| text within fold 3
|}}
| line 3
```

318
The other mode of display is to 'Enter' a fold. When a fold has been entered, it is the only thing displayed on the screen, everything outside this fold is forgotten until this fold is 'Exit'ed. So if the fold marked 'fold within fold' in the example above were entered, the screen would look like this:

```
|[[[ fold within fold
|    Note that this fold has also been brought to
|    the left hand side of the screen. This allows a very
|    big program which has deep indentation to still be
|    comprehensible when displayed on a 80 character wide screen
|]]]
```

Such an editor helps programming top down, modular programs are displayed in a **Top down fashion.** An example would be to look at the display when an occam printer program (for example) is edited: Note that the implementation detail is completely hidden, and that the overall program structure is clearly displayed.

```
PROC Print ( CHAN Keyboard, Screen, To.Filer, From.Filer )
... Constants
... Procedures
... Variables
SEQ
... find out which file to print --> FileName
... find out which printer --> PageWidth,Length, PrinterType,...
PAR
  FileStreamer(To.Filer,From.Filer, FileName, StreamToFormat)
  Formatter(StreamToFormat, PageWidth,PageLength, StreamToFormat)
  CommandsToControlCodes(Formatter,PrinterType,PrinterOut)
```

7. **AN EXAMPLE OF AN OCCAM PROCESS**

There are several stages to the design of an occam process and the same sequence can be followed at each level of the design. It is always difficult to follow this sequence rigorously, but it this more or less what happens.

1. I/O specification
2. Drawing pictures of subprocesses
3. Express the algorithm in structured English
4. Repeat this for all subprocesses
5. Implement

The pictures of subprocesses map very easily onto occam processes. If one uses boxes to represent the for identifiable tasks and use lines running between them to representing the flow of data, then the boxes become processes and the lines channels. Moreover, the definition of the I/O and the structured English forms the basis of the comment fold inserted at the top of each process.

The example given here attempts to represent the processes and communication involved in giving a lecture. It is by no means complete and has never been implemented, but should nonetheless give some idea of how easy it is to design distributed algorithms in occam.

Looking at the input to this system, there are the number of participants, a lecturer, some lecture notes and slides. The output is a possible change in the internal state of the participants. At the top level the simulation will show the way
in which the lecturer and the participants interact. The lecturer reads the notes, gives both audio and visual output and gets feedback by both listening and watching. A participant might listen, watch and from time to time give some feedback. The first attempt at a visual description of these interactions, including the audio and visual systems, might look like this:

The top level process in this simulation reflects the pictorial structure of the diagram and we represent the boxes by processes of the same name and dataflow lines by channels of the same name:

```
PROC LectureSimulation ( VAL INT NoOfParticipants,
    VAL [ ] BYTE LectureNotes )
    [ NoOfParticipants ] CHAN MicToPart, ProjToPart, PartToFeed:
    CHAN AudioToMic, VisToProj, FeedToLect:
    SEQ
        ... Check that NoOfParticipants > 0      [ hidden in a fold ]
        PAR
            Lecturer ( FeedToLect, AudioToMic, VisToProj, LectureNotes )
            Microphone ( AudioToMic, MicToPart )
            Projector ( VisToProj, ProjToPart )
            Feedback ( PartToFeed, FeedToLect )
        PAR pt = 0 FOR NoOfParticipants
            Participant( MicToPart[pt], ProjToPart[pt], PartToFeed[pt] )
```

The next stage in the development of this program is to write the various procedures in 'Structured English'. In this case we have used occam structures as the basis for this pseudocode and folds are used to hide detail. The procedures 'Lecturer', 'Participant' and 'Microphone' are presented below (the other two procedures are similar to 'Microphone').

320
PROC Lecturer (CHAN feedback, audioout, visualout, VAL []BYTE Notes)
  declaration
  ... initialise
  WHILE (NOT AtEndOfLectureNotes)
  PRI ALT
  feedback ? Question
  ... try to answer question
  TRUE & SKIP -- no input ready on Feed channel
  SEQ
  ... read next line of notes
  IF
  line = NewPage
  ... put new slide on projector
  line = text
  ... read line
  TRUE
  SKIP -- do nothing
  ... any more questions?
  ... output 'finished'
  ... accept any feedback (clapping??)

PROC Participant (CHAN listen, watch, feedback)
  definition
  ... initialise
  WHILE (NOT TheEnd)
  SEQ
  ALT
  listen ? Token
  ... input rest of sentence in to a buffer
  watch ? Token
  ... read rest of line into a buffer
  IF (Asleep AND (Token = EndOfLecture))
  ... wake up, TheEnd := TRUE
  Asleep
  SKIP -- do nothing
  TRUE -- otherwise
  SEQ
  ... try to assimilate information
  ... take action as necessary
  ... clap for time depending on interest and comprehension
  ... finish off

PROC Microphone (CHAN In, []CHAN Out)
  definition
  SEQ
  Going := TRUE
  WHILE Going
  SEQ
  In ? Message
  PAR pt = 0 FOR (SIZE Out)
  Out[ pt ] ! Message
  IF ...
  message was 'finished'
  Going := FALSE
  TRUE
  SKIP

8. GENERAL STRATEGIES FOR PARALLELISING PROBLEMS

In our work at Southampton, we have identified three common types of parallelism [6], which I will call event, geometric and algorithmic. I will describe each one in turn and show how they may be used. It is also possible to implement a mixture of these types of parallelism and this is shown in the final example. It is also useful to define the 'efficiency' of a multi-processor implementation 'e' as:

321
Time taken on one processor (sequential algorithm)

\[ e = \frac{\text{N * Time taken on N processors (distributed algorithm)}}{\text{sequential algorithm}} \]

We will discuss what sort of efficiencies can be achieved in each case.

### 8.1 Event Parallelism

In this type of parallelism the same code is run on each processor, but with different data. The data could be from an external source, or it could be a different seed for a random number generator. This form of parallelism looks superficially like SIMD (Single Instruction Multiple Data) but is not, as each processor can choose to branch to a different part of the program, due to the data dependencies. The only inter-processor communication needed is for the passing on of results. If this is only needed infrequently in comparison with the amount of processing needed, then the efficiency can be high (say 0.9 to 1.0). This is much the easiest type of parallelism to implement.

A typical arrangement for a group of transputers might be:

```
results <----- |<----- |<----- |<----- |<----- |<----- ...
```

| data | data | data |

### 8.2 Geometric Parallelism

In this case the data area is split up amongst the transputers. Again the same code is normally run on each processor. Unlike the 'event' case however, inter-processor communications are necessary during the computation to access data held on a different processor. The efficiency is therefore dependant on the balance of computation to communication, and this in turn depends on the size of the data area assigned to each processor. The number of nearest neighbours that each processor can communicate with also affects the performance. Since the transputer has four links, communicating to more than four nearest neighbours incurs a considerable communication overhead. Efficiencies have been obtained in the range 0.8 to 1.0 using 16 processors. Geometric parallelism is trickier to implement than event parallelism, since both data and results have to be communicated, and a more complicated communications strategy is required.

Two typical arrangements we have implemented on transputers: Firstly, a one dimensional problem with periodic boundary conditions. Note that the lines represent bi-directional communications.
Secondly a two dimensional problem, again with periodic boundary conditions:

memory mapped communication to HOST

8.3 Algorithmic Parallelism

This corresponds to using a Dataflow or Demandflow style of programming. Each process does a small amount of work on the data as it flows though the system. Typically these programs require a more complex control structure, and are consequently more difficult to write and debug. Case studies have indicated that efficiencies of 0.6 to 0.8 can be obtained for a limited number of processors.

An example processor arrangement for algorithmic parallelism involving several 'pipes' of transputers. The arrows indicate the direction of the flow of data:

8.4 Combinations

There are many different ways in which these three types of parallelism can be combined. Two of these seem to be particularly promising, namely using algorithmic within both event and geometric.
Algorithmic within Event

One transputer may not have the speed to deal with a particular data stream so a network of transputers might be used. In the example below a chain of processors is used, in general the algorithm will determine the network:

```
results
HOST <----------+----------+----------+----------+----------+----------...
          |          |          |          |
          |          |          |          |
          |          |          |          |
          |          |          |          |
          |          |          |          |
data       data       data
```

Algorithmic within Geometric

Algorithmic parallelism seems to give reasonable efficiencies for small numbers of processors, but the efficiency will drop as the number of processors being used increases. Geometric parallelism needs a large data area, and direct communication to each of its nearest neighbors (in each dimension of the problem) to achieve high efficiencies. Thus replacing the single transputer in the geometric case by a cluster of transputers in an algorithmic network can work very well. The amount of data stored at a node may now be increased, because the processing of data is speeded up by using more processors. Moreover the number of links connecting nearest neighbors is no longer restricted to four. An example would be the following multi-transputer 'node' to be placed at the corner of a cubic lattice:
9. **CONCLUSIONS**

The transputer hardware performs very well and the T800 will provide a 10 fold increase in floating point performance. The use of T800 transputers in large multi-transputer systems will enable programmers to achieve Gigaflop performance for a fraction of the cost of a second generation supercomputer such as a CRAY-2.

Moreover, it is the provision of such cheap processing power locally, and at the disposal of individual research groups that has the potential to revolutionise the whole of computational science and engineering. Transputers are (or will be) a cheap and natural building block for such systems and occam is a natural language to implement parallelism. The world is a parallel place and to describe it in the most natural way requires a language which supports concurrency.

Who knows, eventually we may even be able to assemble enough computing power to solve QCD (Quantum Chromo Dynamics) - a task which will probably require at least 100 Gigaflop years!

**ACKNOWLEDGMENT**

I wish to acknowledge the help of ESPRIT Project 1085 which provided equipment and promotes the exchange of software and ideas between partners. The Occam Users Group is another forum for such exchanges and also has been the source for much of the information on transputer based machines. I would also like to thank Tony Hey and the 'software team' at Southampton for their comments advice and encouragement.

* * *

**REFERENCES**

1. INMOS Ltd. The transputer reference manual, (INMOS, 1000 Aztec West, Almondsbury, Bristol, UK)
3. May, D. Occam 2 product definition, (INMOS)