COMPUTER SYSTEMS AND SOFTWARE ENGINEERING

Edited by

Patrick Dewilde

and

Joos Vandewalle

SPRINGER SCIENCE+BUSINESS MEDIA, B.V

Computer Systems and Software Engineering

Computer Systems and Software Engineering

State-of-the-art

Edited by

Patrick Dewilde
Technische Universiteit Delft

and

Joos Vandewalle
Katholieke Universiteit Leuven

SPRINGER SCIENCE+BUSINESS MEDIA, B.V.

ISBN 978-1-4613-6555-6 ISBN 978-1-4615-3506-5 (eBook)
DOI 10.1007/978-1-4615-3506-5

Printed on acid-free paper

TABLE OF CONTENTS

February 17, 1992

Editorial: the State of the Art in Computer Science and Software Engineering

At the occasion of the CompEuro'92 Conference in The Hague this spring, we have asked an editorial board consisting of some twelve reputed researchers in computer system design and software engineering in the Benelux to assemble a program of State of the Art lectures covering their entire domain, and in which up to date information would be presented to practicing engineers and researchers who are active in the field and have a need to know. We are proud to present the results of our combined efforts in this book. We truly believe that we have succeeded in our goal and that the present book offers the broad overview we aimed at with contributions that treat all the main topics with elegance and clarity. In addition to State of the Art lectures, also some tutorial as well as keynote lectures have been added, all with the purpose of providing the global coverage desired.

The order in which the topics are presented goes roughly from bottom to top, from low level hardware issues to high level intelligence, from computing to applications. Although many topics are incommensurate, we have opted for the more specific first, the more general later, without any presumption. The first lecture is just as valuable as the last: the topics rise in a kind of spiral. There is an undeniable randomness in the choice, the reader will find important and interesting topics from the first to the last contribution.

We start out with some survey material. W. Proebster gives a knowledgeable survey of the history of memory management and technology. As a long standing IBM scientist (who is now with the University of Munich), he has had an insider's view on the many developments of memory technology. His conclusion is that the sky is the limit and much more will be forthcoming. The second contribution is also from an outstanding IBM design scientist who is now a Professor at the University of California at Berkeley, Bob Brayton. His topic is, of course, logic design, but with a new view on optimization. Since this is one of the main themes in sound design technology, a fresh view on it is particularly welcome. The contribution of Peter Pirsch remains in the realm of hardware design, in particular the design of video signal processing algorithms and architectures. Peter's group in Hannover was one of the very first to study and implement sophisticated high speed dedicated computing systems. He instructs us on how to achieve optimization in a global design space in which software, algorithms and architectures interplay. The last contribution on hardware design is due to Lothar Thiele of the University of Saarland in Saarbruecken. Lothar is also very much concerned about the connection of algorithms and architectures, especially in the area of massive but nearly regular or piecewise regular computations. He presents a systematic methodology for the design trajectory that carries the designer from concept to architectural description, all within one framework.

Moving one level upwards, or if you wish, one spiral turn further, we reach three papers that carry the central term 'parallelism'. The first in the series is devoted to a type of neural networks called cellular neural nets, which may be viewed as a locally distributed neural computer. It yields the promise of a much larger array than possible with the classical multilayer type. The two following papers cover two most important issues in parallel processing: that of programming and that of load-balancing. R. H. Perrott reviews in his paper the various possibilities in parallel programming languages and how they have developed. In contrast to sequential computing where a single underlying architectural model, the von Neumann machine, is often tacitly assumed, there are many possible architectural models available for parallel computing. Is unity in variety possible? That is the question addressed by Perrot. The paper of Dirk Roose and coauthors brings us back to the theme 'operations research', in their paper dedicated to parallel computing on distributed memory parallel computers. There could possibly not be a more central paper in this conference since it merges two of the three central themes. Therefore, it appears in the center of the book as well...

The following paper has also a central quality but of a different nature. It brings us another turn further on our high winding spiral. Operations research has very much to do with strategic decision analysis. Decisions are not taken in a vacuum, they originate from group dynamics in a management team. Simon French approaches these issues from an information systems background. It is refreshing to see how on a macroscopic field problems arise that are similar to those on the much more restricted field of software engineering, and how these problems may be approached in an analogous way.

This brings us to one of the central themes in software engineering: reasoning and logic programming. Three lectures are devoted to that topic. The first, by Y. Kodratoff gives a survey of ten years of advances in machine learning. Methods for machine learning are built on three kinds of possible inferences: deductive, inductive and analogical. Kodratoff describes different methods that have been created during the last decade to improve the way machines can learn, using these inference techniques. In the second paper, J.W. Lloyd of the University of Bristol dives directly into issues of the design of logic programming languages, their declarative semantics and the software engineering support needed for them. Raghu Ramakrishnan and coauthors treat in their contribution the evaluation problem of logic programs. Here also, optimization is a key issue, efficiency of computations, memory utilization, improvement of query handling.

Engineering nowadays requires more than hard nosed i'rule of the thumbi'-ing. Already we have seen that mathematical principles underlie modern principles of declarative parallel programming. Modern signal processing is based on sophisticated functional analysis. The construction of a high speed, high density VLSI computing system is not possible without the use of highly optimized circuit construction software which again is based on deep insights in combinational mathematics. So what about software engineering? Kit Lester states *our methods of* *software construction are mainly intuitive and generally have an air of "string and chewing gum" construction. If we are truly to* Engineer *programs, we instead need mathematically-based methods either of constructing the programs, or of verifying intuitively-constructed programs.* In his paper he treats not only correctness of program source code or program specification, but goes further and takes the discussion all the way to the executable program.

The next-to-last three papers in this volume are devoted to a topic for which a complete State of the Art book could be produced: visualization. At the system's level there is the question of how an application designer could construct a man-machine interface. Jan van den Bos of the Erasmus University in Rotterdam presents a system that allows designers to construct the essential parts of a graphical men-computer interface, the presentation, the dynamic aspects and the coupling to application functions. A beautiful application of visualization principles is offered by volumetric medical image data representation. The state of the art in that field is presented by Zuiderveld and Viergever, who pay special attention to strategies that improve image generation speed. Techniques to improve the quality of the image are also covered in that paper. J.J. van Wijk of ECN closes the sequence of papers on visualization and gives an overview of methods by which the results of large scale simulations and measurements can be presented. The division of the visualization process in different steps, the interaction with the user, the use of hardware and the different software possibilities are all considered.

The closing paper is in another direction of application and treats an important problem in medical informatics: that of picture archiving and its connection to communication. The sheer mass of information, the need to store it in an accessible way, and the necessity to communicate that information induce a discipline in its own right which is surveyed by Rudy Mattheus of the Hospital of the Free University of Brussels.

The editorial board for this book consisted of the following persons: E. Aarts and F. Lootsma (Optimization and Operations Research), P. Vitanyi (Theory of Computing), H. Sips (Parallel Processing), L. De Raedt and Y. Willems (Machine Learning and Knowledge Acquisition), J. ter Bekke (Databases), A. Bultheel and B. De Moor (Numerical Mathematics), M. Bruynooghe (Computational Logic), M. De Soete (Data Security), F.W. Jansen (Computer Graphics), J. van Katwijk and K. De Vlaminck (Software Engineering and Compiler Construction), P. Suetens (Medical Computing and Imaging). Our thanks go to all of them, for helping us out in selecting lecturers, for advising us on the program and for requesting the contributions. Our very warm thanks go to all the authors who have written such beautiful papers and are introducing us so effectively to the State of the Art in Computer Systems and Software Engineering.

Delft, January 1992
Patrick Dewilde and Joos Vandewalle.

The Evolution of Data Memory and Storage: An Overview

Walter E. Proebster

Technische Universität München,
Institut für Informatik, Orleansstr. 34, München, Germany

Abstract. An overview is given on the development history of the key technologies of data memory and storage. For each of them the essential characteristics for their application in computer systems and also their relation to competing – preceeding or replacing – technologies is described. Comparisons of characteristic values of speed and packaging densities along with a list of historical milestones are added. As a conclusion this overview extending over many decades shows that the progress of this field, which is of dominating importance for system design and application, has not yet reached saturation of progress now and for many years to come.

1. The Role of Data Memory and Storage

From the very beginning of data processing, data memory and storage has always played a dominant role in the architecture, the design, the implementation and the operation of data processing systems, be it that we regard a system of the historical past, of the early times of electronic computers, or of our times.

The economic importance of data memory and storage is substantial: Almost one third of the world-wide yearly production value of the total systems hardware is devoted to this sector. The demand of computer users for more data memory and storage at improved cost/performance can still not be satisfied, even with the enormous investments in research, development and industry in the past decades and today.

The term *"memory"* is used for random access devices, such as semiconductor memories, the term *"storage"* for sequential access devices, mostly electromechanical devices such as disc and tape storage. For reasons of simplification, the term "memory" will be mostly used in the following for both classes.

2. Characterization, Storage and Memory hierarchy

Memory and storage units can be characterized mainly by the following three criteria:

— capacity, in bits, Bytes, kBytes(10^3 Bytes), MBytes (10^6 Bytes), GigaBytes (10^9 Bytes), TeraBytes (10^{12} Bytes),

— access and cycle time of a memory/storage cell for writing or reading its content.

— cost or price related to bits.

The large variety of memory technologies naturally leads to enormous differences of capacities, speeds and cost. These differences, to a large extent, influence and determine the application of the various memory classes within a dataprocessing system and, most important, its proper structure:

For fast memories, the costs per bit are high; as a consequence, only small capacities can be realized economically. In contrast, slow memories generally offer low costs per bit and therefore allow the realization of large capacities. These considerations lead to different memory levels embedded in a hierarchical structure: fast memories for registers and smaller buffers (caches), slower random access devices for main memories, fast sequential access devices for back-up storage, slow ones for archival storage.

Based on the voluminous literature, one can state without exaggeration that almost all fields of electrotechnology, physics, chemistry and material science have been examined for suitable memory phenomena. Due to the great importance of memory for data processing, physicists, engineers in research, development and industry have been engaged for many decades now to explore, realize and improve new ways and methods of storing data.

3. Memory Evolution and Impacts on and from other technical Disciplines

Figure 1 shows an overview of the most important and interesting memory technologies, the time of their conception, their peak of application and partly also the time of their replacement by other, superior technologies.

There are 5 time periods to be distinguished:

1. From ancient times to the end of medieval times with mechanical devices.
2. The time of electromechanical devices up to about 1950.
3. The period of pioneer computers — Zuse, ENIAC, Princeton: IAS — where communication technology provided valuable components for the construction of computer memories such as relays, cross-bar switches, etc.
4. The time of maturity of data processing systems, where reciprocally strong positive impulses were exercised from computer memory technology to communication technology leading e.g. to electronic telephone exchanges.
5. Our present time, where computer memories profit considerably from developments of the entertainment industry such as the compact disc: CD and the digital audio tape: DAT technology.

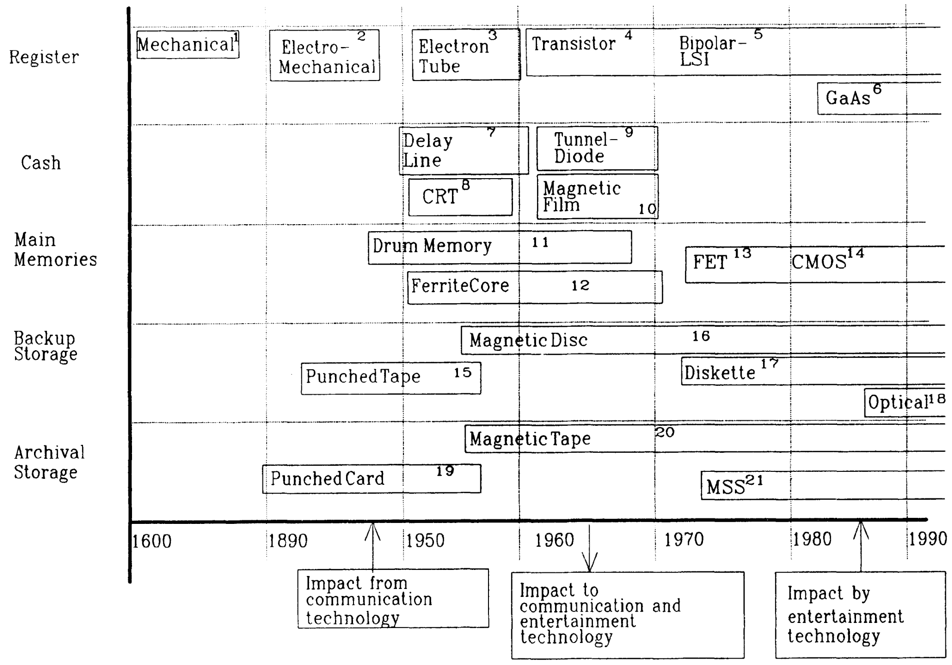


Fig. 1. Overview of historical evolution of the major memory and storage technologies

For all replacement processes of new technologies against established ones it can be observed that the only real chance of success of any new technology lies in offering substantial and not just incremental improvements over the established technology. Effects impeding or retarding the replacement process are: confidence in the established technology, human expertise, process methods of development and manufacturing, risks of the new technology: unknown technical problems, "Not invented here" (NIH), high investments in manufacturing, and, last but not least, problems of compatibility of hardware and software of new technology with the established one.

Due to these problems, many new approaches which looked promising in the beginning did not or not yet succeed despite of many years of intense research and development efforts, e.g. ferroelectric memories, superconducting memories, multivalued memories, biological storage which will not be treated in the following.

A great number of milestones exists for data memory and storage, some of the important ones are shown in Fig.2. A number of them will be discussed in the next sections.

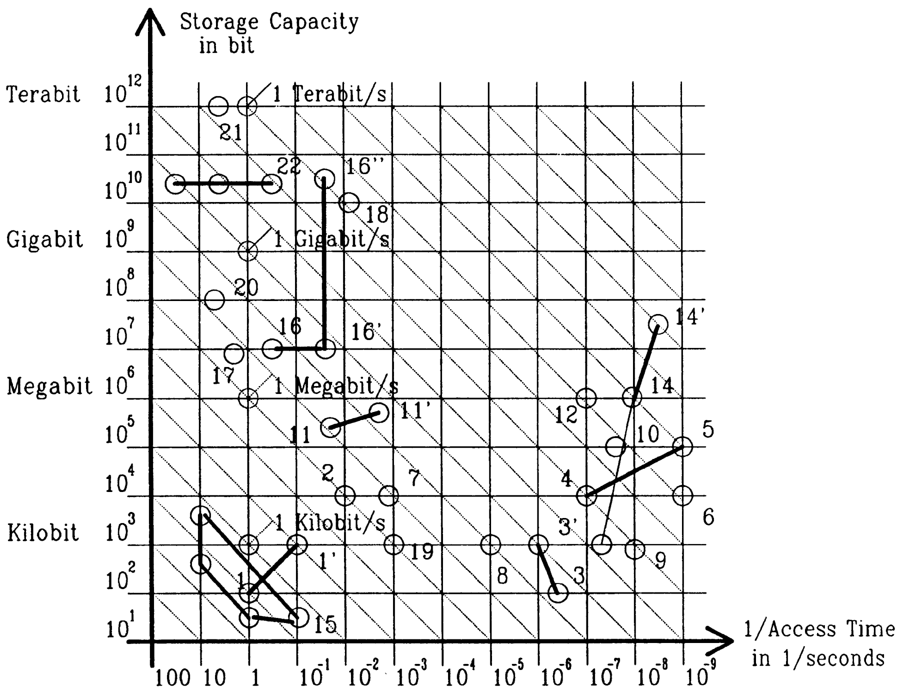
1350	Use of pins in cylindrical drums (mechanical drum storage) for the control of chimes and musical mechanisms.
1728	The French mechanic Falcon employs wooden boards containing holes at certain places for initiating control functions in looms.
1805	Joseph-Marie Jacquard uses big cardboards with holes for the control of the looms later to be named after him.
1870	Wheatstone prints the output of transmitted morse signs on paper strips.
1882	Herman Hollerith starts working with punch card machines. The punch cards have initially 12 lines and 20 columns.
1890	Electromechanical counters working on a decimal basis are used for the storage of data. As a general rule, a decimal unit was made up as replaceable element.
1900	Western Union (USA) develops a 5–track teleprinter paper strip with punch and reader devices.
1901	Ch. Maurain describes the magnetical characteristics of thin iron layers.
1919	W.H Eccles and F.W. Jordan develop a bistable switching element with two electronic tubes (now called flip-flop).
1928	Pfleumer of Dresden, Germany, obtains a patent on his magnetic tape storage.
1933	G. Tauschek of Vienna, Austria, applies for a patent on cylindrical storage.
1937	K. Zuse develops an electromagnetical binary storage unit. It consists of two layers of crossed, thin metal plates and metal pins at the crossing points, each of which can be placed in two positions.
1937	Punched paper tapes are introduced as data- and program storage medium in computers.
1938	K. Zuse, without prior knowledge of H. Aiken's work, uses relays as storage elements.
1943	Dirks sen. and jun. develop the concept of today's drum storage.
1946	Electron tube flip-flops are first used in the ENIAC-computer by I.P. Eckert and J.W. Mauchly.
1946	First magnetic drum storage units developed by Booth at Birkbeck College, London, and by Billing at Max Planck Institut, Göttingen.
1947	F.C. Williams (Manchester) develops a method for storage of binary values on the screen of a cathode ray tube. Cycle time 8μs, 1024 bits per access.
1947	Mercury delay line storage is introduced for fast memories in computers.
1948	Development of ultrasonic quartz memories.
1950	H. Aiken's first use of magnetic tape storage in a digital computer (Mark III).
1951	J.M. Forrester publishes on ferrite core memory and the principle of coincidence addressing.
1953	First use of ferrite core memory in data processing units. First use of Ferrite-Kemel-Storage in data processing units.
1955	The IBM laboratories in San Jose, California (Goddard, et al.), develop the magnetic disc storage, a new type of storage with high capacity and fast access, which is first used in the RAMAC-machine.
1955	H.O. Leilich develops a drum storage for PERM: 600,000 bits, 250 revolutions/s, 200 magnetic read/write heads.
1956	D.A. Buck describes the first low-temperature switch, called Kryotron. Various application investigations for memory applications in computers ever since.
1956	Development of magnetostrictive ultrasonic delay line for data memory units.
1957	Basic investigations by Tensor, Gianola and Long on magnetic wire storage with thin films.
1958	Delivery of the first transistorized computers with flip-flop memories.

1959	The Bell-laboratories develop a photographic storage with 2^{10}bits and $5\mu s$ access time, with cathode ray tube, optical lense arrangement photographic plates and photocell.
1959	IBM develops a photographic storage on rotating glass discs as storage for a language translation system.
1965	Development of the "chain-store", a fast thin film memory element by IBM (USA).
1966	Start of systematic research on semiconductor monolithic integrated circuits with high integration density.
1966	Univac, Bull and Nippon Electric use magnetic ultrasonic delay lines for the first time.
1967	IBM develops a 120 ns-cycle time memory unit with thin magnetic films and 600,000 bits capacity.
1969	First use of semiconductor memories in a data processing unit: IBM announces model 360/85 with monolithic buffer storage of a capacity of up to 32,000 bit.
1970	ILLIAC IV-Prototype: first computer unit constructed with monolithic main storage.
1971	RCA announces a prototype of a semiconductor read/write memory with a capacity of 1 million bits.
1980	First production of semiconductor memories with a capacity of 64 kbit/chip.
1983	Semiconductor memories with a capacity of 256 kbit/chip.
1986	Semiconductor memories with a capacity of 1 Mbit/chip.
1987	The first prototype of an optical "Write Once Read Mostly" (WORM) hard disc are announced.
1990	Semiconductor memories reach a capacity of 4 Mbit/chip.
1991	Hitachi announces its first prototype of a 64 MBit/chip.

Fig. 2. Milestones in research and development of data memory and storage

For typical realizations of the various memory technologies shown in Fig. 1, capacities and the inverse of the access-time are plotted in Fig. 3. The product of both values yields a figure of merit: the larger the capacity and the shorter the access time, the higher is the quality of the memory. Significant improvements of these values expressed in bits/sec can clearly be recognized from one memory generation to the next one.

For comparison, estimated values of the human brain are also entered in this figure. Research of the structure and operation of the human brain has progressed remarkably in the last decades. We know for example almost certainly that our brain consists of about 30 billion nerve cells, and that it is structured in subunits, similar to a data processing system, with short, medium and long access times.



1 Abakus
1' Mechanical
2 Electro-Mechanical
3 Electron Tube-Register
3' SAGE- Register
4 Discrete Transistor
5 Bipolar- LSI
7 Delay Line
8 CRT-Storage
9 Tunnel Diode
10 Magnetic Film
11 Drum Memory "Manchester"
11' Drum Memory "PERM"
12 Ferrite Core Storage
13 FET - Storage
14 CMOS 1Mbit
14' CMOS 64 MBit Storage (Hitachi)
15 Punched Tape
16 Magnetic Disc
17 Diskette
18 Optical Storage
19 Punched Card
20 Magnetic Tape
21 MSS
22 Human Brain

The product of both values is a measure of quality

Fig. 3. "Figure of merit" of data memory and storage technologies

Area density and volume density are plotted in Fig. 4a) and 4b), for the time span from 1600 to 1991 . Also, these figures illustrate convincingly the rapid advances of memory and storage technology.

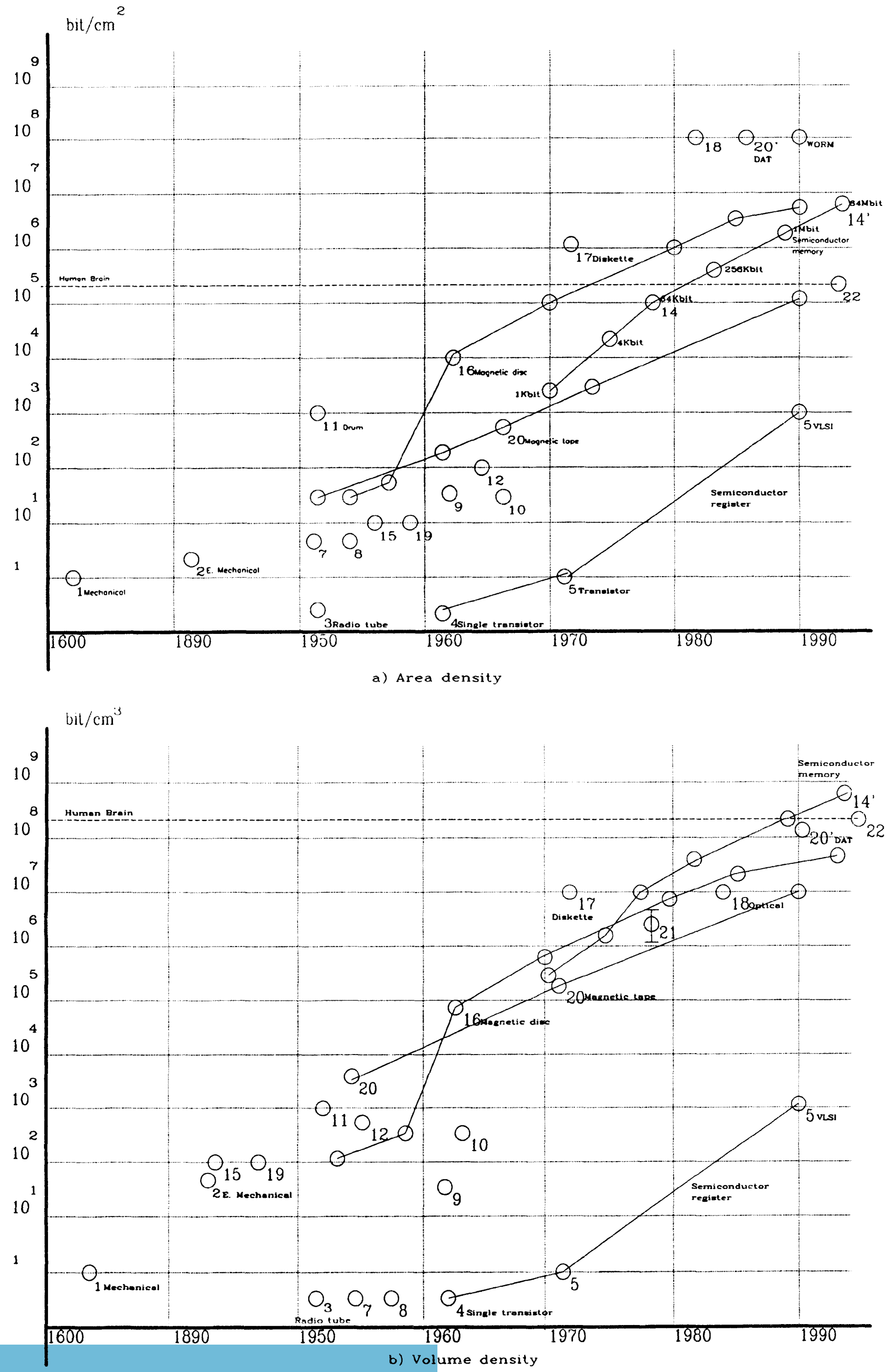


Fig. 4. Historical development of area density (a) and volume density (b)

Two classes of memories with special functions will be treated only very briefly; namely read-only memories, ROM, and associative memories.

In *read-only memories*, the information to be stored can be entered only once, mainly at the manufacturing location of the memory component or of the computer system, only in rare cases by the system user. The elimination of write operations by the user allows the realization of memory units with lower cost per bit, shorter access times and larger capacities. Furthermore, the information is stored permanently and cannot be destroyed inadvertently by the user or by the interruption of power. In addition, read-only realization of new memory technologies accomplish in many cases sooner technical feasibility and therefore reach sooner the market place.

Associative memories, which search for the content of memory cells and not according to a specified address, constitute a combination of memory and logic.

In the following sections, the evolution of memory and storage technologies and products will be sketched briefly, subdivided in:

1. Registers and caches,
2. Main memories,
3. Back-up
4. Archival storage.

Registers and Caches

As first realizations of memory units, we can consider the counting stones of past cultures - e.g. of the Mayas - and the abacus of some 2000 years ago. Capacities of the abacus seldom exceed some 100 bits, access times, given by the speed to move our fingers, are in the range of one second.

The first calculators, invented by Schickard, Hahn, Pascal, Leibniz and others, employed mechanical storage devices, wheels and drums with sticks. Capacities and access times again ranged around 10 digits and one second respectively (Fig. 5).

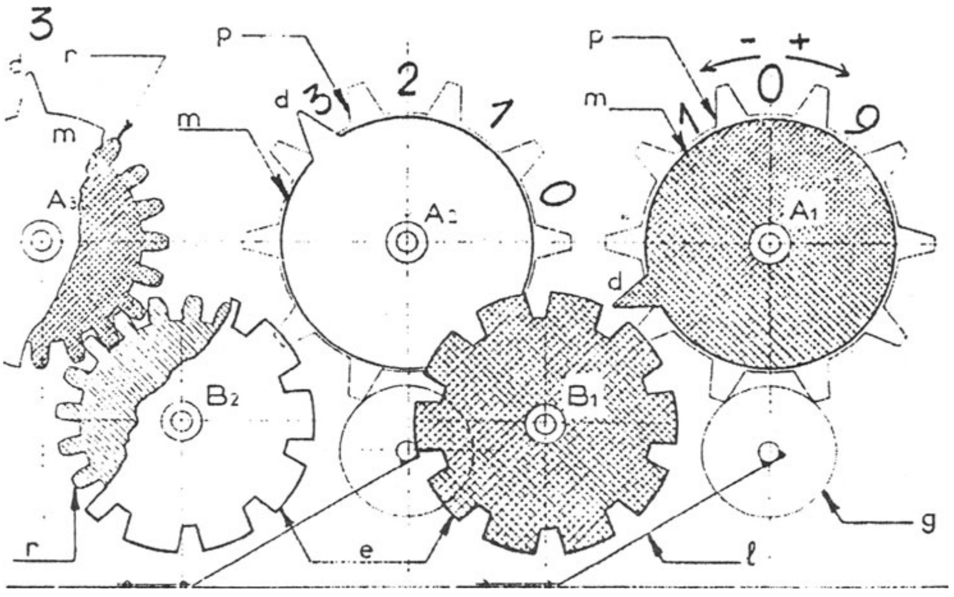


Fig. 5. Gears of Wilhelm Schickard

The electromechanical technology, from the end of the last century up to about 1960, was the basis of electric calculators and electric computers, e.g. for the punched card calculating computing system of Herman Hollerith, and even for the first computers controlled by programs stored in read/write memories around 1940. In comparison to mechanical storage units, capacities and access times could be increased by a factor of 10 and more.

The electrical relay, Strowger – and crossbar switches – allowed the realization of several thousand bits with access times of about 10 milliseconds. The first completed computers of Konrad Zuse were built with this technology.

After 1940, electromechanical elements were replaced by electronic components: for registers, vacuum tube flip-flops, with switching times around one microsecond, for caches, cathode-ray tubes (CRT), utilizing charge storage: the famous "Williams Tube" and acoustic delay lines using mercury tanks with piezoelectric generators and sensors (Fig. 6), later magneto-strictive wires with electromagnetic transducers and sensors.

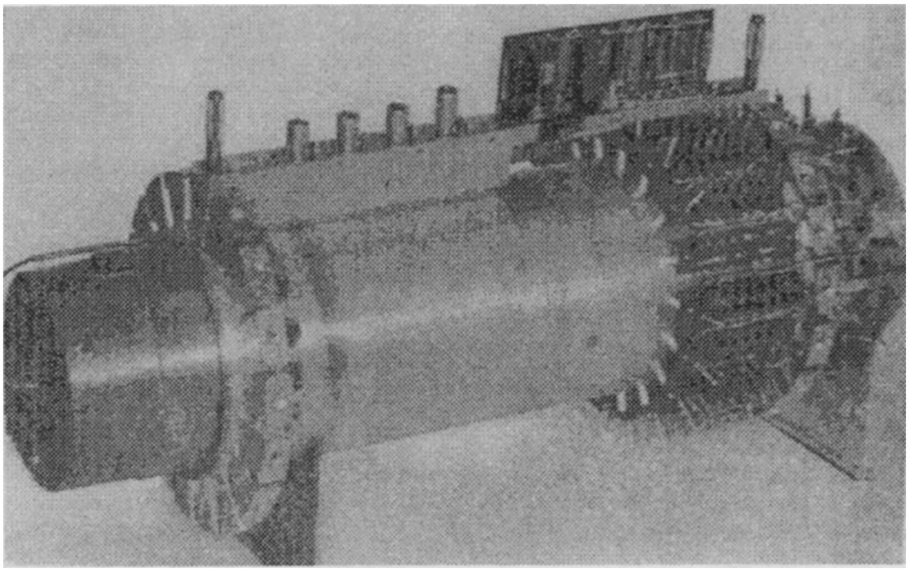


Fig. 6. Acoustic delay line of UNIVAC I with a capacity of 18 × 480 Bit and an access time of 222μs

From about 1965 semiconductor memories replaced all other technologies for registers and caches.

For conventional applications, semiconductor registers and caches are integrated with the logical elements of processors in one chip, to reduce cost and increase reliability.

Gallium arsenide memory chips may, in the future, gain increased importance for extremely short access times in supercomputers, based on expected improvements of the gallium arsenide material's technology, which is more complex and less understood than the silicon material technology.

Main Memories

Around 1950, drum memories were used as main memories in many pioneer computers. Their capacities seldom exceeded one million bits, the access times were limited by the rotational speed of the drum – material stability, centrifugal forces – to the range of milliseconds.

As an example, the drum memory of the PERM is shown in Fig. 7 with a capacity of 400,000 bits and a mean access time of two milliseconds, resulting from the very high rotational speed of the drum with 15,000 revolutions per minute.

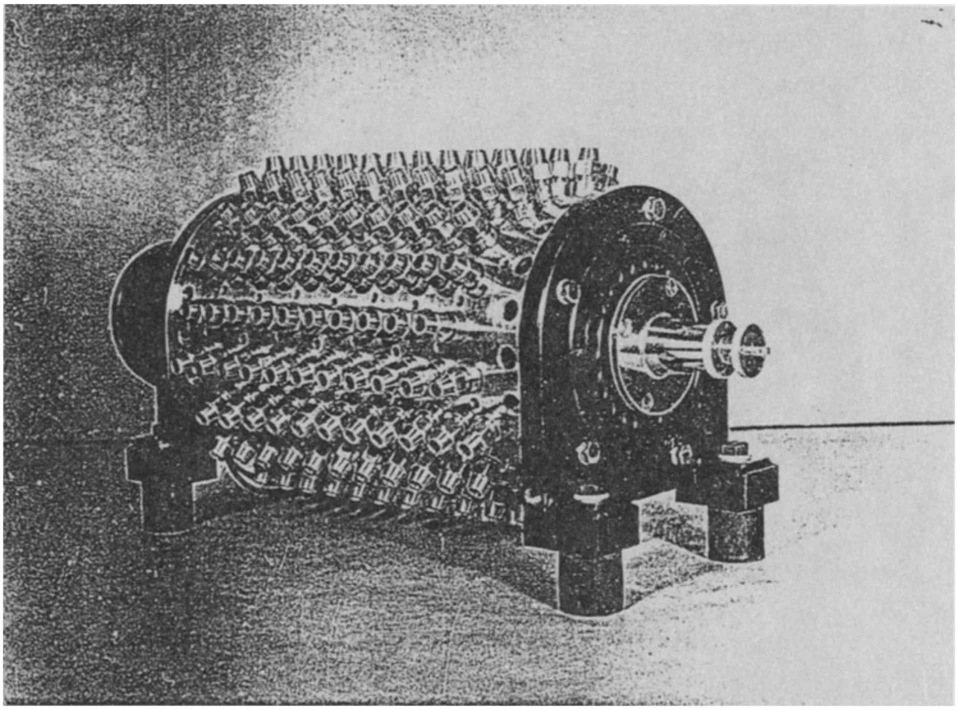


Fig. 7. Drum memory of the PERM

Up from about 1955, drum memories were replaced by ferrite-core memories, which offered faster access and larger capacities: switching times were typically around one microsecond, capacities up to one megabyte and more. The second mayor advantage of ferrite-core memories is their random access property: the access time of the drum memory – and also of all other magneto-mechanical and of many optical memories – is dependent on the actual distance of the requested memory cell relative to the read/write station. In contrast, the access times of ferrite-core memories – and for almost all other solid state memories – are practically independent of the address of the requested cell.

The addressing cost of ferrite-core matrices – first with vacuum tube circuits, later with transistors – could be contained by multiple coincidence schemes based on the static and dynamic rectangular hysteresis loop of the ferrite cores.

The cost of production of ferrite core matrices could be reduced dramatically by ingenious automatic wiring techniques as compared to wiring by hand, which was used in the beginning. The ferrite cores, which could be manufactured at low cost, were put in order by vibration in the grooves of a metal plate. After fixation of these cores by a grooved counter plate, all isolated x and y wires were inserted in one manufacturing pass through the holes of the ferrite cores, which finally had only an outer diameter of 0.3mm (12 mils) and an inner diameter of 0.175mm (7 mils) (Fig. 8).

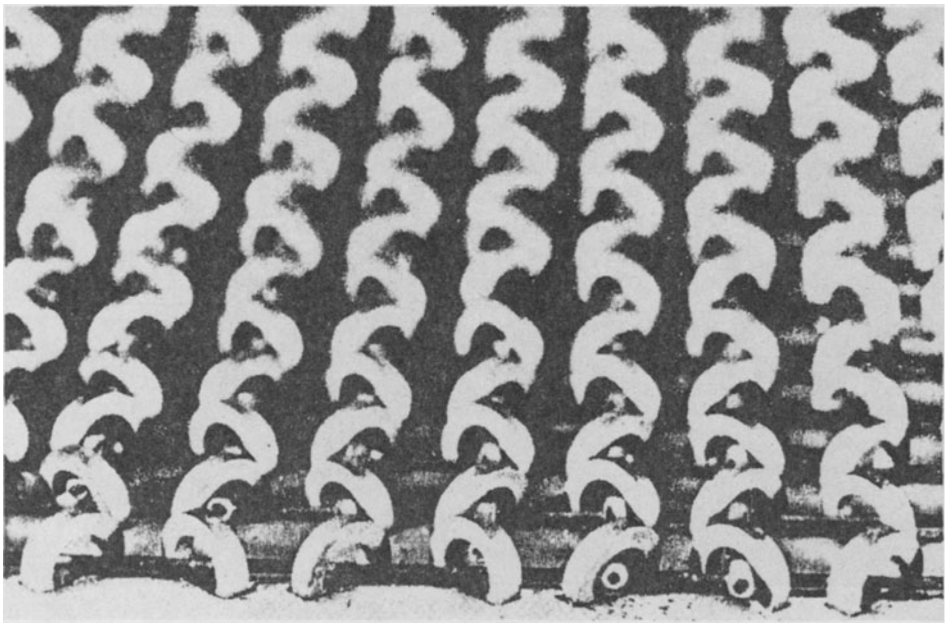


Fig. 8. Ferrite cores positioned in core loader matrix cavities

A further reduction of the access and cycle times by a factor of 100 or more was possible by thin magnetic films. The rotation of magnetization in thin magnetic films allowed to obtain switching times of about 1 nanosecond. Thin magnetic film memories with an access time of 60 nanoseconds, a cycle time of 120 nanoseconds and a capacity of 600,000 bits have been developed and manufactured for many years by IBM, employed in the supercomputers System 360/95. Two disadvantages are, however, inherent with all magnetic memories as compared to semiconductor memories, which caused in the end also their defeat in the fields of buffer and main memories: firstly, the transformation of the representation of information from its electrical to the magnetic form and vice versa is very inefficient: practically only a conversion efficiency of 10^{-3} to 10^{-5} can be reached. Secondly, if the size of a memory cell is reduced, the relation of the energy of the read signal to that of the disturb signal is not constant but it is decreased significantly.

The competition between magnetics and semiconductors in the fields of buffer and main memories in the beginning of the 60ies was carried out by both parties with tremendous effort. The outcome of this competition is well-known: in a very short time span, semiconductor memory elements were generated, initially with only a few bits per chip, soon up to one megabit per chip, and recently even 64 megabits per chip, with chip dimensions of originally only of one by one millimeter, up to 20 by 20 millimeters today.

Semiconductor memories can be produced at low cost and high performance by a combination of high-resolution photolithography, oxidation, etching and diffusion technologies with silicon, the best-known element of the world today.

To increase the capacity, the semiconductor memory cells can be reduced in size by three methods:

1. Improved lithographical resolution by transition from long light wave length to short light wave length, possibly to electron beam and even to x-ray lithography,
2. Simplification of the memory cell from initially 6 transistors per cell, later on to 4, and finally to 1 transistor per cell, combined with capacitive charge storage,
3. Refined packaging technology of the memory cell. Examples are here the arrangement of the storage capacitance of dynamic random access memories (DRAMs) in vertical grooves and furthermore the use of multilayer semiconductor memory structures.

The experts on magnetism defeated by the experts on semiconductors in the field of main memories withdrew to the exploration of new memory materials and concepts: Improvement of materials for disc and tape storage and also the exploration and development of non-volatile memories, memories with a special function which at that time the still young discipline of semiconductor memories could not fulfil.

Without special precaution the stored information is lost in semiconductor memories when the power supply is switched off. In many cases this is particularly disturbing, for example when it is necessary to secure the state of a system. To overcome this problem, but also to develop a product in-between memory and storage, namely to bridge the gap in access times between the magneto-mechanic storage devices and the faster random access solid state memories, for almost one decade physicists and engineers have worked in Europe, the USA and also in Far East countries in research and development on magnetic bubble memories.

New magnetic materials on the basis of metals of rare earth have been found suitable to store information by magnetic domains. In these materials and arrangements the vector of magnetization is perpendicular to the plane of the storage medium in contrast to conventional arrangements, for which the vector of the magnetization is in the plane of the magnetic layer. These materials have a very high coercitive force which is required here to withstand the strong demagnetization field. In Fig. 9 the creation of magnetic cylindrical domains is shown which are briefly called "bubbles" and which form the elements of binary storage. These elements with a diameter in the order of microns, can be generated, moved in the film plane, duplicated, merged and erased by controlling external magnetic fields. As an example of a memory configuration the classical minor/major loop arrangement of the Bell-laboratories is shown in Fig. 10. Bubble memories with capacities of 256,000 bits per chip with access time in the millisecond range can be produced. Though the technical feasibility was proven and in many places not only samples of bubble memories and computers were produced, bubble memories have not been employed in conventional computers as in the meantime also non-volatile semiconductors have been developed based on the insolated FET-controlling gate, briefly called floating gate and this at a smaller cost per bit and with shorter access times. An exception is the application of magnetic bubble memories for military devices due to their stronger resistance against alpha rays in comparison to semiconductor memories.

The development of the magnetic bubble technology influenced positively the advances of the magnetic and optical disc storage.

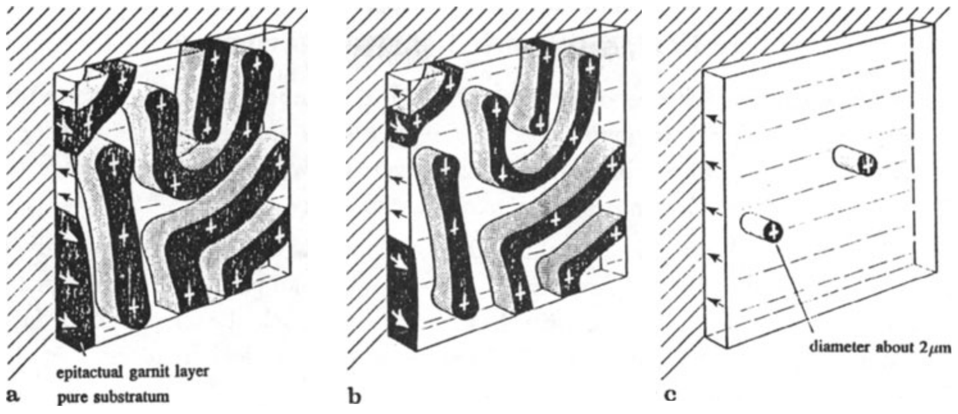


Fig. 9. Generation of magnetic bubbles

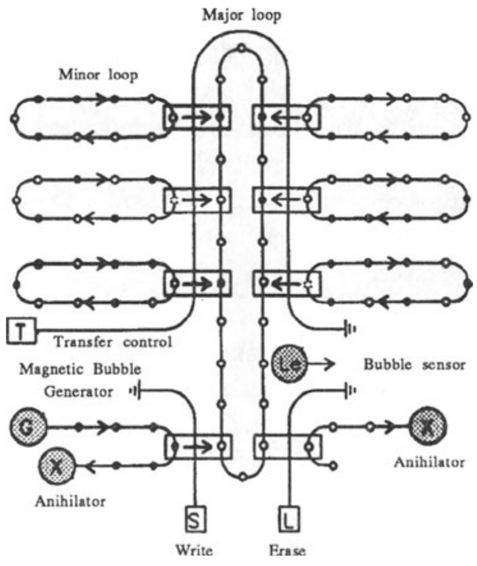


Fig. 10. Shift register storage with magnetic bubbles

Backup storage: Magnetic and optical disc and diskette storage

For backup storage already around 1955 the magnetic disc storage started to replace the magnetic drum mainly because of its higher storage volume density – essentially enabling three-dimensional storage versus two-dimensional storage on the magnetic drum – leading to larger storage capacities.

The strive for high area and volume density is one of the predominant goals in research and development of new storage technologies and products. For magnetic disc memories, where the read/write head must not touch the magnetic surface in order to secure high reliability and lifetime, one basic design rule is that the following three form factors:

— pole gap,
— distance between read/write pole and recording surface and
— the thickness of the magnetic recording layer

should be about equal and as small as possible.

The reduction of the distance between magnetic head and recording layer through the last decades is shown in Fig. 12. This was only possible by improved engineering designs and manufacturing methods for the magnetic heads, their mountings and of the magnetic recording layer itself. The demand of high bit density in combination with short access times leads necessarily to very high data rates which in turn demand excellent high frequency properties of the read/write heads and also of the recording layer.

These basic considerations illustrate the development history of magneto-motoric storage units – drum, disc, and tape storage: For the magnetic read/write head the first embodiments, for example in the magnetic drum of the PERM, consisted of nickel/iron alloys. Later on ferrite read/write heads offered improved high frequency properties, better manufacturability and lower cost. Today ferrite read/write heads are again replaced by arrangements with thin magnetic films – deposited electrolytically, by sputtering or by evaporation – which allowed to reach values of 1 micrometer and below for pole gap, distance between head and recording layer and thickness of the recording layer itself. The more complex and more costly production process of thin magnetic films is more than compensated not only by the small dimensions of the storage cell, but also by the higher density of the magnetic flux and the improved high frequency properties.

A similar development can be observed for the magnetic recording material. In the first drum memories nickel wire was wound spirally around the drum surface and served as carrier of information. It was soon replaced by iron oxide and ferrite, well proven and improved in many products through many decades. Presently, ferrite is again being replaced by thin hard-magnetic metal layers, which yield higher bit densities.

The development of magnetic disc storage units started with constructions of only one read/write head for many recording discs (Fig. 11). They were followed by arrangements with one read/write head per recording surface placed on comb shaped access mechanisms along with replaceable stacks of magnetic discs. This feature, to mount and dismount disc stacks to and from a magnetic disc storage unit which was highly desirable from a user's point had to be abandoned very soon as with increased bit densities and the resulting tight tolerances head assembly and disc pack have to be matched.

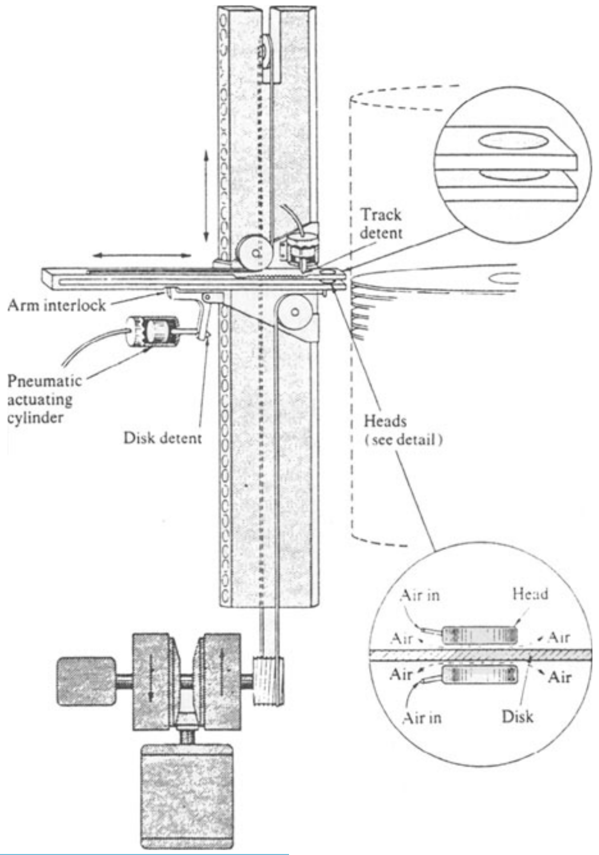


Fig. 11. Carriage and actuator of the IBM 350 magnetic disc storage unit

Technical Data	IBM 350 1957	IBM 3380 Model E 1985
Capacity in MBytes	5	5000
Number of drives	1	2
Number of discs	50	2x9
Access time in ms	600	17
Transfer speed in kBytes/s	8.8	3000
Head-to-surface-distance (10^{-3}mm)	31.5	0.3
Pole gap (10^{-3}mm)	39.4	0.6
Thickness of magnetic layer (10^{-3}mm)	47.0	0.6
Track density (SP/mm)	0.8	63.0
Bit density (Bits/mm)	3.9	600.0
Data density (Bits/mm^2)	3.1	37800.0

Fig.12. Evolution of magnetic disc performance

Because of the steadily reduced flying heights of magnetic heads over the recording surface special constructions were required: flying heights in the range of one micron and less can only be reached by a balance of spring forces and aero dynamical forces which act on the read/write head carrier, sometimes also combined with separate start and landing tracks without information content for the start and stop operation of the disc stack.

The diskette store is the small brother of the magnetic disc storage. It was developed around 1970 with its low cost plastic base for the magnetic layers at relaxed characteristic values for personal computer applications. Here, the magnetic head is allowed to touch the storage layer, which is much thicker and which is not so frequently accessed as for the magnetic disc storage. Capacities are today usually in the range of about one MByte per diskette. In research laboratories with new coding methods and with magneto-resistive read heads storage capacities up to 200 MBytes can be reached with access times of only a few milliseconds.

The development of optical data storage was initially pushed by the entertainment electronic industry. The compact disc technology, mainly developed by Philips and Sony served also as basis for the optical storage of digital information. Pits of different length in the range of micrometers carry the information. Using semiconductor lasers the information can be read without mechanical contact of the read head. The capacity of optical discs with diameters of up to 30 centimeters is in the range of two GBytes with access times of only few milliseconds.

The main disadvantage of the first optical discs, namely the lack of the write function will be overcome with new technical approaches. Of the different proposals particularly the magnetic version deserves our attention. Here, the storage cell has a preferred magnetic axis perpendicular to the recording plane similar as in the magnetic bubble cell. Information is written by heating the cell by a controlled laser beam above the magnetic Curie point and cooling it in a magnetic field perpendicular to the recording surface. The magnetic Faraday or Kerr effect is used for reading.

This latest development shows clearly the technological direction for backup storage: the technologies of the optical storage disc and that of the magnetic storage disc begin to merge whereby the advantages of both technologies are being utilized: adequate distances of read/write head to the recording layer, on one hand, reliable read/write function on the other hand and also the possibility to exchange the information carrier. This demonstrates a rare example of harmonic cooperation between two different technical disciplines.

Archival storage

Punched card files can be considered the first archival storage technology. Huge databases emerged from the onset of our century. In 1965, when electronic dataprocessing was already fully developed, the yearly production of punched cards reached its peak. The punched card has, with 80 columns of 12 positions each, a capacity of around 1,000 bits. At a processing speed of about one thousand cards per minute, the mean access time was in a range between milliseconds and seconds.

At the very beginning of electronic data processing, the application of magnetic tape units – initially conceived for the recording of audio signals – started for low-cost storage of large data volumes. Due to the translatory motion of the tape from the storage reel to the machine reel, the access time for the storage cell is not constant but depends on its position relative to the read/write head at the start of a read cycle.

Of the many proposed configurations of magnetic tape units, three closely related forms have prevailed until today, which have been remarkably improved in the last years with respect to capacity and access time.

The first type is the classical magnetic tape unit with storage reels – the so-called long-tape-unit – with eight to nine parallel information tracks (Fig. 13). By major and minor tape loops and tape drives with very small inertia, the start/stop acceleration could be increased up to 500 times the gravitational acceleration. For a typical tape length of 800 meters and a bit density of 250 bit per millimeter, the total capacity of the magnetic tape is about 100 megabytes, access time is, at a tape velocity of up to 5 meters per second, in the range of seconds.

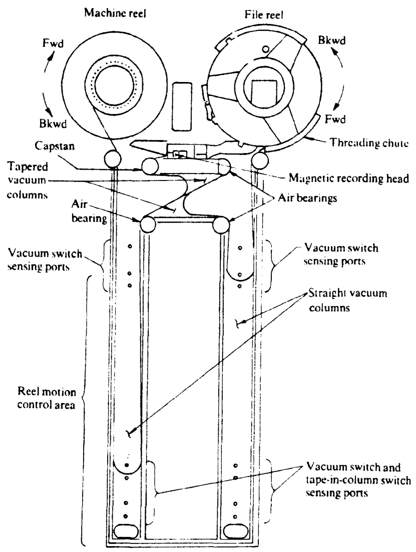


Fig. 13. Tape path for the IBM 2420 model 7 magnetic tape storage

A substantial disadvantage of these constructions is the necessity that the tape reels and later the tape cassettes have to be loaded manually. In an attempt to overcome this drawback, a mass-storage system with magnetic tape cassettes has been developed around 1970 by IBM with automatic loading of 5,000 cassettes by robots to two read/write stations. Each cassette has a capacity of 50 megabytes, stored on an about 8-cm-wide magnetic tape which could be written and read by rotating magnetic heads tilted 45 degrees relative to the tape drive. Access and storage of each of the 5,000 cassettes between the two read/write stations and the honeycomb storage wings was accomplished within a few seconds. Quite a number of these mass storage systems have been manufactured and installed. Due to problems of software support and compatibility with other storage systems, this product line has been abandoned already for several years.

Today advanced magnetic tape and cassette storage units with loading and unloading by robots reach capacities of up to thousands of Terabytes and access times in the range of minutes

The advances of magnetic disc storage resulting in increased capacities and reduced access times, caused that the start/stop-operation of the magnetic tape units just described was no longer necessary. As the second type of magnetic tape units, for archiving of databases, today the so-called streaming mode is therefore used almost exclusively, in which data is transferred to and from magnetic disc units with continuous tape motion.

Tape units with 18 magnetic tracks and corresponding magnetic heads have been developed especially for this mode of operation. Recently even improvements to 36 magnetic tracks and magnetic heads have been announced.

An even further increase in storage density is offered by the third kind of magnetic tape units originally developed by the entertainment electronic industry, the digital audio tape (DAT), video tape recorder with tilted recording and rotating magnetic head. The storage density can be increased by a factor of 10 as compared to the tape unit with tape reels or tape cassettes of the first kind, mainly because of overlapping tracks at the write operation. In this mode, no unused magnetic areas on the tape are left between the individual tracks. Cross-talk between the tracks is reduced by pairs of read and write heads which are tilted - analogous to the operation of a stereo-gramophone-disc - by 40 degrees against each other.Today DAT data cassettes are available with a capacity 5 gigabytes and more and access times in the range of minutes.

In all likelihood, the magnetic tape technology will persist and not be replaced by their competitors - essentially the magnetic disc and the optical disc - mainly because of its extremely low cost per bit resulting from the low cost, replaceable data carriers.

4. Conclusion

Throughout all these decades of memory developments covered in this paper, many deep-rooted discussions about the future of data processing and especially about the future of data memory and storage took place. And often the conviction was expressed that in foreseeable future a retardation of the progress and a consolidation process would occur. Each time, such predictions were wrong: as shown here, even today we do not notice any retardation of the development dynamics. The physical limits of important present memory and storage technologies are known, but many years of intensive research and development activities are necessary to approach these limits. In addition, new ideas will emerge to improve the characteristics of memory and storage – capacity, access time, cost – and to explore new ways and methods, and – last but not least – to obtain functional improvements by the integration of memory and logic operations.

5. References

BIL 77 H. Billing, "Zur Entwicklungsgeschichte der digitalen Speicher". *Elektron. Rechenanlagen 19*, p. 213–217, May 1977

BOB 75 A. Bobeck et al., "Magnetic Bubbles: An emerging new Memory Technology", *Proc. IEEE*, vol 63, p. 1176, 1975

BOW 77 D. Bowers, "Floppy Disk Drives and Systems", *Mini-Micro Syst. 10*, p. 36, 1977

CHA 78 H. Chang, "Magnetic-Bubble Memory Technology", *Marcel Dekker*, 1978

COU 70 E.D. Councill, et al., "A 275-Nanosecond Coincident Current Ferrite Memory", Digest of the *Intermag Conference Washington DC*, April 1970

DIE 60 W. Dietrich, W.E. Proebster, "Nanosecond Switching in Thin Magnetic Films", *IBM Journal of Research and Development 4*, p. 189, 1960

ECC 19 W.H. Eccles, "A Trigger Relay Utilizing Three-Electrode Thermionic Vacuum Tubes", *The Radio Review 1*, p. 143-146, 1919

ENS 48 A.G. Enslie, et al., "Ultrasonic Delay Lines", *J. Franklin Institute 245*, p. 101-115, 1948

ESC 80 A.H. Eschenfelder, "Magnetic Bubble Technology", *Springer*, 1980

FLA 63 J.-P. Flad, "Les trois premieres machines a calculer", Schickard (1623), Pascal (1642), Leibniz (1673), Universite de Paris, June 1963

FOL 30/4 O. Folberth, H. Bleher, "Grenzen der digitalen Halbleitertechnik", *ntz*, Bd. 30, Heft 4

FOR 51 J.W. Forrester, "Digital Information Storage in Three Dimensions Using Magnetic Cores", *J.App.Phys. 22*, p. 44, 1951

GAN 75 K. Ganzhorn, W. Walter, "Geschichte der Datenverarbeitung", *Springer*, 1975

GIT 89 W. Gitt, "Information - Die dritte Grundgröße neben Materie und Energie", *Siemens-Zeitschrift*, 63.Jahrgang, Heft 4, Juli/August 1989

HAR 85 M. Hartmann et al., "Erasable magneto-optical recording", *Philips Tech. Rev.*, Vol. 42, No. 2, p. 37, August 1985

HARK 81 J.M. Harker et al., "A Quarter Century of Disk File Innovation", *IBM Journal of Research and Development*, p. 677, September 1981

HARR 81 J.P. Harris et al., "Innovations in the design of magnetic tape subsystems", *IBM Journal of Research and Development*, Vol. 25, No. 5, p. 691, September 1981

HIL 75 W. Hilberg (Ed.), "Elektronische digitale Speicher", *Oldenbourg*, 1975

KAU 73 H. Kaufmann (Ed.), "Daten-Speicher", *Oldenbourg*, 1973

LEN 78 E. Lennemann, "Tape libraries with automatic reel transport, digital memory and storage", *Vieweg*, p. 65, 1978

KEY 75	R.W. Keyes, "Physical Limits in Digital Electronics", *Proc. IEEE*, Vol 63, May 1975
KRY 86	M.H. Kryder, "The Special Section on Magnetic Information Storage Technology", *Proc. IEEE*, Vol. 74, No. 11, p. 1475, November 1986
LEI 66	H.O. Leilich, "The Chain - A New Magnetic Film Memory Device", *J. Appl. Phys. Vol 37*, 1361 - 1362, March 1966
LEI 89	H.O. Leilich (Ed.), "Tutorial Sessions, VLSI and Computer Periphals", *3rd Annual European Computer Conference*, May, 1989
PHI 81	W.B. Phillips et al., "Innovations in the Design of Magnetic Tape Subsystems", *IBM Journal of Research and Development*, p. 691–699, September 1981
PRO 78	W.E. Proebster (Ed.), "Digital Memory and Storage", *Vieweg*, 1978
PRO 87	W.E. Proebster, "Peripherie von Informationssystemen", *Springer*, 1987
PUG 67	E.W. Pugh, et al., "Device and Array Design for a 120-Nanosecond Magnetic Film Main Memory", *IBM Journal of Research and Development 11*, p. 169-178, 1967
RAJ 52	J.A. Rajchman: "Static Magnetic Matrix Memory and Switching Circuits", *RCA Rev. 13*, p. 183-201, 1952
RUS 67	L.A. Russell, R.M. Whalen, H.O. Leilich, "Ferrite Memory Systems", *IBM Technical Report*, TR 00.1640, August 14, 1967
SCH 89	W. Schilz, C. Butterworth, "VLSI, External Memories and Storage", Digest of *3rd European Computer Conference*, Proceedings VLSI and Computer Periphals, W.E. Proebster, Hans Reiner (Ed.), May, 1989
SIM 65	Q.W. Simkins, "A High-speed Thin Film Memory: Its Design and Development", Proceedings of the *Fall Joint Computer Conference*, 1965
SÖL 78	S. Söll, J.-H. Kirchner, "Digitale Speicher", *Vogel*, 1978
STE 81	L.D. Stevens, "The evolution of magnetic storage", *IBM Journal of Research and Development*, Vol. 25, No.5, p. 663, 1981
WIL 49	F.C. Williams, et al., "A Storage System for the Use with Binary Digital Computers", *Proc. IEEE 96/2*, #81, p.183-200, 1949
WIL 85	M.R. Williams, "Some Anecdotes from the History of Computing - Early Memory Devices -", in *Überblicke Informationsverarbeitung*, Hermann Maurer (ed.), *B.I. Wissenschaftsverlag*, p.22-233, 1985
WIN 85	E.O. Winkelmann, "Ein Vierteljahrhundert Magnetplattenspeicher in der IBM Deutschland", *Datentechnik im Wandel*, (W.E. Proebster Ed.), *Springer*, 1985

OPTIMIZING LOGIC FOR SPEED, SIZE, AND TESTABILITY

ROBERT K. BRAYTON and ALEXANDER SALDANHA
University of California - Berkeley CA

Abstract. Three primary objectives, optimized at all levels of design of integrated circuits are speed, size, and testability. While techniques for individually optimizing each of these are well formulated, until recently not much was known about the interactions between these criteria. There are two important reasons why this interaction is interesting. First, it was unknown whether a tradeoff between the criteria necessarily exists, *i.e.* must the optimality of one objective be sacrificed when the others are optimized? Second, the impact of the tradeoff between the speed, size, or testability on the resulting quality of the optimized integrated circuit was not well understood. This paper surveys the various interactions that are known to exist between these three criteria in combinational logic circuits; most of the results apply to sequential circuits as well. After a review of techniques for optimizing speed, size, and testability individually, the interactions between these criteria are explored with emphasis on recent results relating performance and testability in optimized circuits. A long range goal is to understand the optimum tradeoff (or Pareto) surface in the speed-size-testability space. We wish to devise transformations on logic which are guaranteed to move from one point to another on this surface. This and other open questions are discussed in the final part of the paper.

1. Introduction

Three major axes used to measure the goodness of a logic design are its speed, size, and testability. In this paper, we discuss optimizations done mostly at the technology independent level and use technology independent measures for all criteria. Speed is measured as the longest sensitizable path of the circuit (thus excluding false paths). Size is measured by the number of literals in the logic equations. Testability is measured by the percentage of either the stuck-faults or path delay-faults that can be tested.

Logic synthesis has traditionally addressed each of these parameters separately and more recently, in pairs. Results precisely relating all three are still missing. A long range goal is to understand the optimum tradeoff (or Pareto) surface in the speed-area-testability space. We wish to devise transformations on logic which are guaranteed to move from one point to another on this surface.

In the logic synthesis area, the most mature results optimize for area. Minimal area can be obtained by either redundancy removal or logic minimization using don't cares [8]. Synthesis for 100% testability for stuck-faults can be accomplished either by redundancy removal, or by starting with a two-level circuit and restricting the type of transformations used. Synthesis for speed can be done by transformations such as tree-height reduction, partial collapsing, generalized bypass and select transformations, and technology mapping [41; 45; 29; 6; 44].

In considering pairs of these criteria, the relation between speed and area is somewhat understood since most speed-up algorithms selectively duplicate logic, thus adding area to obtain an incremental speed up. Stuck-fault testability can be directly related to area since an untestable connection can be removed, thus simultaneously reducing area and improving testability. The relation between speed and testability is addressed by the KMS transform which replaces a redundant circuit (for stuck-faults) with an irredundant one with no loss of speed [23]. Recently, the efficiency of the KMS transformation has been improved significantly (requiring only a few seconds on even the largest circuits) by making it a single pass operation on the circuit, during which all false paths greater than the longest true path are removed [33]. This leads to the question of whether it is ever necessary to synthesize circuits with long false paths.

Another set of questions concerns delay testability. Since fast circuits are of interest, it is important that these can be delay tested. Various delay testability conditions have been proposed. Most imply regular stuck-fault testability, but are much stronger. An open question is to find the weakest practical delay testability criteria for which full delay testability can be synthesized. Can we do this without sacrificing delay? How much area penalty must be paid?

At the moment, there are no results yet that relate all three criteria in a tight way. Some open questions are concerned with how nonoptimal the area becomes when the other two parameters are optimized, whether by speed-up transformations or testability optimization (stuck-faults or path delay-faults).

This paper provides a survey of the state-of-the-art in logic optimization for the three criteria discussed above. While all known interactions between the criteria are covered, the focus is on recent results and the open issues that need to be addressed with respect to relating all three parameters. The paper is organized as follows. Section 2 is a review of traditional logic synthesis algorithms that focus on optimizing one of the speed, size, or testability of a design. Section 3 considers the interactions between pairs of criteria. The interaction between performance and testability motivates the KMS algorithm, which is studied in detail in Section 4. Section 5 illustrates the tradeoff surface that exists between all three criteria. Section 6 discusses some open questions concerning delay, area, and testability, and Section 7 concludes.

2. Logic Optimization Criteria

Given a functional description of a system that includes memory constructs, combinational logic synthesis extracts only the combinational portion of the logic for optimization. The memory elements are connected back into the final optimized circuit at the end of the process. [8] is a complete description of the algorithms and approaches used in this level of synthesis. In this section, the three most common goals of any logic optimization problem are considered.

2.1. Area optimization

By far the best understood aspect of combinational logic synthesis is the manipulation of logic equations to yield an implementation of minimal area. When the target technology is a two-level implementation both exact and heuristic algorithms are well established [13; 7]. In two-level implementations, the area of an implementation is proportional to the number of terms. A secondary function is the number of occurrences of the variables (referred to as the number of literals). In multilevel logic, exact minimization algorithms are much harder to achieve, since the solution space is considerably larger due to increased degrees of freedom compared to two-level logic [8]. However, several techniques exist that yield sufficiently high-quality area-minimal solutions. In a multilevel implementation, area is most often estimated by the number of literals in the implementation. For multilevel implementations two basic approaches are adopted. The first is a rule-based approach consisting of the application of selected transformations from a given collection of rules developed by experienced circuit designers [14; 4]. The second is algorithmic based [10; 2]. Many industrial synthesis systems employ the second technique followed by the first.

A successful strategy employed in the latter approach is to decompose the process into two steps: technology independent optimization and technology dependent optimization. This often simplifies the design and optimization problem to be solved, while still yielding an efficient solution. Technology independent operations, which apply to generic gates independent of technology-specific information, may be further classified as algebraic or Boolean. While algebraic operations restrict the set of operations used to optimize circuit structure, they are fast; they can be performed in polynomial time in the number of variables of a function [46]. Although Boolean operations are more time-consuming, they are essential in obtaining minimal circuits [36]. The technology dependent optimizations consist of *mapping* the generic gates to a specific library of cells, corresponding to a target technology [16].

2.2. Performance optimization

Performance optimization is often the primary optimization criteria in logic designs [1; 8] (subject to some area constraints). A typical problem is to improve the delay of an existing circuit structure. At the technology independent level this is done by incremental modifications to the network topology to yield a *faster* circuit. Three transformations [41; 29; 45], that have recently matured into efficient and feasible algorithms to reduce delay are considered in this section. At the technology dependent level, delay may also be minimized instead of area during the *mapping* phase [44]. An alternate technique, often used in conjunction with mapping, is the insertion of buffer cells with high capacitative-drive properties to further improve delays through gates propagating signals to several different parts of the circuit [5].

Timing optimization is viewed as a three-phase process [8]. In the first phase, global restructuring is performed on the circuit to reduce the maximum level or the longest path in the circuit. Typically, this is accomplished in a technology independent fashion. For example, changing from a ripple-carry-adder to a carry-look-ahead adder (or something in between) is accomplished in this phase. The second phase is dependent on the target technology and is referred to as *technology* *mapping*. Techniques for reducing the delay of mapped circuits, with minimal area increases, have been studied recently in [44]. The final phase is to speed up the circuit during the physical design process. Transistor sizing [20] or timing driven placement of modules [31] are examples of such optimizations. In this phase, when an actual design exists, a more accurate timing analyzer is used to fine-tune the circuit parameters. In this paper, only the first phase of timing optimization is considered, *viz.* technology independent logic resynthesis. The other technology dependent optimizations of buffering [40] and transistor sizing [20] do not change the testability of a design.

2.2.1. The delay of a circuit

Before describing performance optimization techniques it is necessary to define the metric used for estimating the performance of a circuit. Early logic synthesis systems used the number of levels of logic as a crude estimate of performance. This estimate is easily made more accurate by considering technology dependent delays. However, in some designs the longest paths can never contribute to the delay of the circuit. These paths are termed *false paths* and should not be considered in estimating the delay [28]. In this paper, performance is measured by the length of the longest *sensitizable* path. Informally stated, a path P is sensitizable if there exists an input vector such that the signal value (that propagates) along P determines the value at the output of the circuit. See [27] for a complete treatment of this subject. Here it is mentioned that recent advances [30; 19] have allowed the efficient determination of the longest sensitizable in large benchmark circuits.

2.2.2. Tree-height reduction

The first approach to technology-independent timing optimization is tree-height reduction [24; 15; 41]. The algorithm of [41] uses a timing driven decomposition of the network into 2-input gates. This is important since the manner in which a complex gate is implemented changes its delay characteristics. Various models are used for computing delays. One is a fast technology mapping [16] of the two input gates into a standard cell library. This provides more accuracy to the timing estimates. An algorithm, based on timing constraints, for decomposing a complex function into two input gates is also used. This is done recursively from the bottom up, so that at each stage the input arrival times are fairly accurate, and thus subsequent decomposition of the upper nodes can be based on these updated arrival times.

The resynthesis algorithm takes as input a network of 2-input NAND gates and inverters. Timing constraints are specified as the *arrival times* at the primary inputs and *required times* at the primary outputs. The algorithm manipulates the network until the timing constraints are satisfied or no further decrease in the delay is possible. The output of the algorithm is also in terms of 2-input NAND gates and inverters.

An outline of the timing optimization algorithm is given in Figure 1. Details of each step of the algorithm and its parameters are in [41]. Given the primary input arrival times, the arrival times for each of the signals is computed. Using the required times at the outputs, the required times for all signals are computed. The *slack* at a node s, is defined as $R_s - A_s$ where A_s is its arrival time and R_s its required time. An *ϵ-network* is defined as the sub-network containing all signals with slack within ϵ of the most negative slack. The procedure *node_cutset(ϵ-network)* determines a cut-set [21] of nodes, each of which must be sped up in order to realize some global delay reduction. The procedure *partial-collapse(n, distance)* collapses all the nodes in the transitive fanin of the node *n* which are themselves in the ϵ-network and at most distance *distance* from n. All such internal nodes that fanout elsewhere will be duplicated in this process (thus delay is improved at the expense in area). The *speedup_node(n)* procedure performs timing driven decomposition on the complex node *n*. The overall strategy attempts to place late-arriving signals closer to the output. Thus, in the first phase, divisors (or common sub-expressions [9]) of the node with early arriving inputs are factored out. After all such divisors are exhausted, a timing based decomposition into a 2-input NAND gate tree structure is done. This routine again places late-arriving signals closer to the output and also ensures that the final timing-optimized network is composed of 2-input gates only. A final step of area recovery may be performed to merge identical gates.

Figure 2 illustrates the procedure on an example circuit. Primary input e is a late-arriving signal which causes output n to be available after 6 units of time. Since this signal is critical the timing optimization procedure attempts to reduce the delay to n. In the first phase the ϵ-critical network (encircled in the first schematic of the figure) is identified. In the second step, these nodes are collapsed to yield a complex node at n. Note that nodes h, j and m are duplicated since they are required at other places in the network. Finally, the large node n is decomposed based on arrival times of its inputs. Notice that e passes through fewer gates since it arrives after the other signals. The final optimized delay for n is 4 units. The new delay of the circuit is now 5 units, and the critical path is the path from e through m. While a simple unit-delay model is used in this example, several different delay models can be employed to improve the correspondence of the delay estimate on the 2-input NAND gate network with its final mapped implementation [41].

2.2.3. *Generalized bypass and select transforms*

Two recent techniques that have generalized the approaches used in deriving the carry-skip adder [25] and the carry-select adder [42] from a ripple-carry adder are discussed in this section.

Generalized bypass transform: The carry-skip adder is derived from a conventional ripple-carry adder and provides a substantial improvement in performance

```
/* $\eta$ is the Boolean network to be speeded up.
$distance$ is the number of levels up to which the critical
fanins are collapsed. */
speed_up($\eta$, $distance$) {
    do {
        delay_trace($\eta$).
        $\epsilon$-$network$ = select_critical_network($\eta$, $distance$).
        $node\_list$ = node_cutset($\epsilon$-$network$).
        Foreach node $n \in node\_list$ {
            partial_collapse($n$, $distance$).
        }
        Foreach node $n \in$ node_list {
            speedup_node($n$).
        }
    } while (delay($\eta$) decreases &&
            timing constraints not satisfied).
}
```

Fig. 1. Outline of the timing optimization process

at a very small expense in area. This is achieved with the addition of logic that bypasses the long carry-chain through the original ripple-carry adder [25]. In other words, speedup is achieved by converting a long (sensitizable) path in the ripple-carry adder to a false path in the carry-skip adder [32]. A technique that generalizes this to speed up arbitrary circuits by making critical paths false is described in [29].

This approach is illustrated using Figure 3. The speed-up is accomplished by making the long path from f_m a false path. Note that the condition under which the value of the node g is sensitized to the value of node f_m is specified by the Boolean difference term $\frac{\partial g}{\partial f} = g_f \oplus g_{\overline{f}}$. On connecting the Boolean difference to the select input of the MUX, the output is the same as f_m when the Boolean difference is true[1], but is independent of f_m (which is computed by the original circuit) when it is false. Under all conditions the long path from f_m through g is never sensitized, *i.e.* it is a false path.

Generalized select transform: Another type of fast adder is a select adder (or conditional-sum adder) [42]. A select adder is obtained from a ripple-carry adder

[1] In some cases f_m might have to be inverted to obtain the correct output.

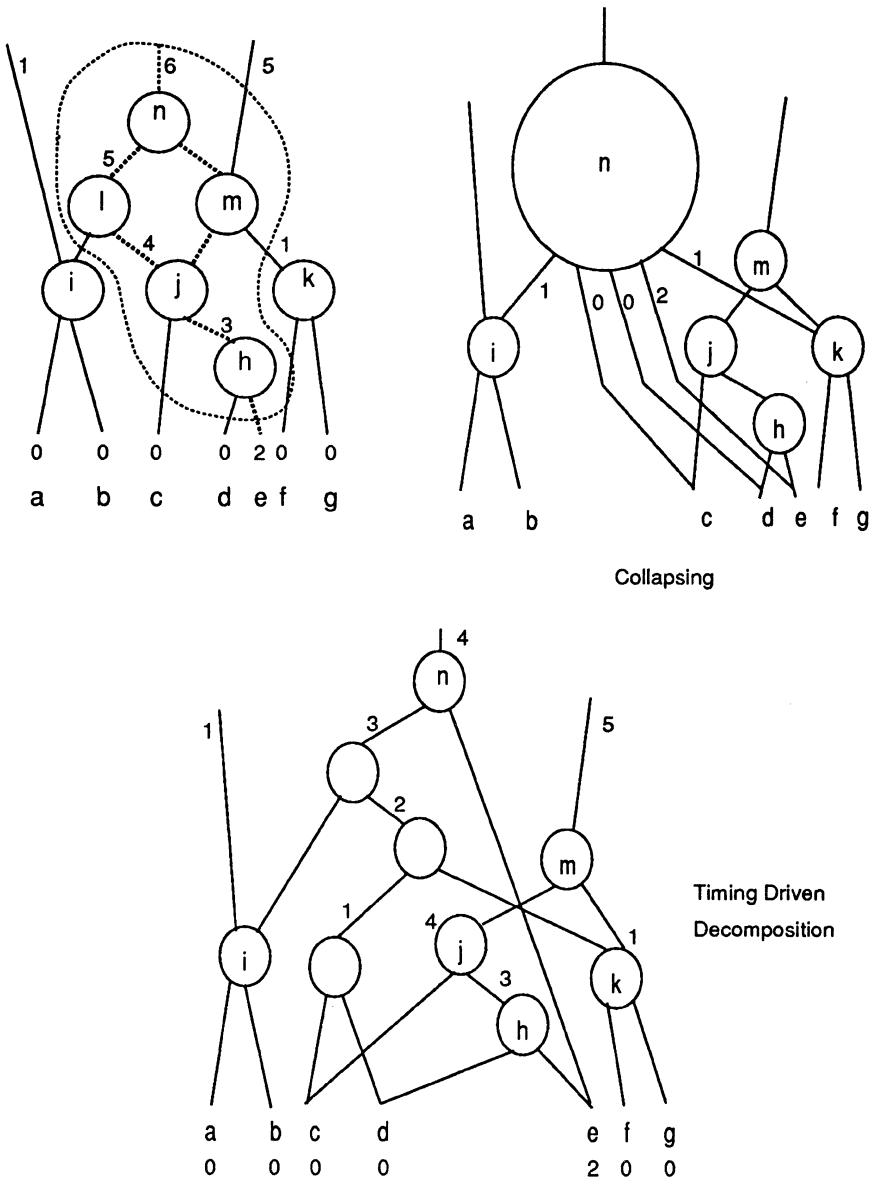


Fig. 2. Example: Timing optimization process

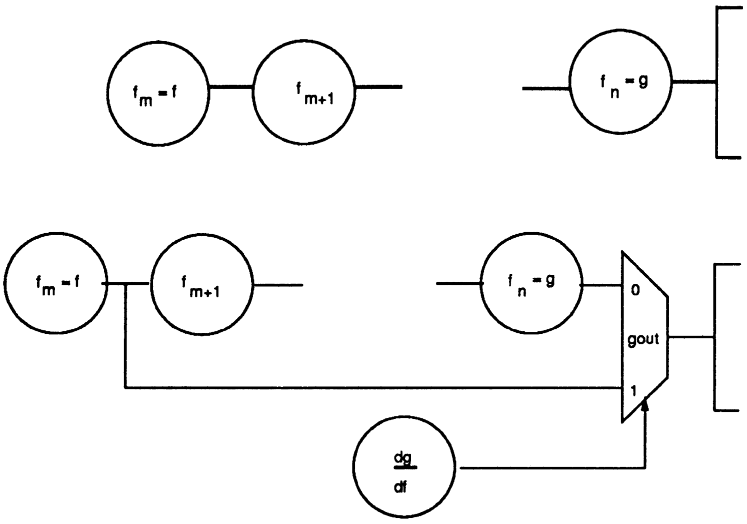


Fig. 3. Generalized bypass transform

by moving the critical carry signal very close to the primary outputs. This is done via the Shannon cofactor operation [39]. A generalization of the Shannon cofactor is effective in delay optimization of circuits with only a few long critical paths. This technique is reported in [6] and is illustrated using Figure 4, where the bold path is critical. The transformation moves the late arriving signal *a* nearer to the output.

The question arises as to how the two transformations must be applied in a network. Both the generalized bypass and generalized select transform can use the same strategy as shown in Figure 1. The two steps of *partial_collapse* and *speed_node* are replaced by the transformations described above.

2.2.4. *Clustering and partial collapsing*

A third approach to technology independent delay optimization is provided in [45]. The algorithm works in two steps. The first step performs a partial collapse of the circuit to obtain nodes with large functions. This is done using a well known clustering algorithm but driven by delay estimates provided by a technology mapper. The second step factors and simplifies the circuit without increasing the number

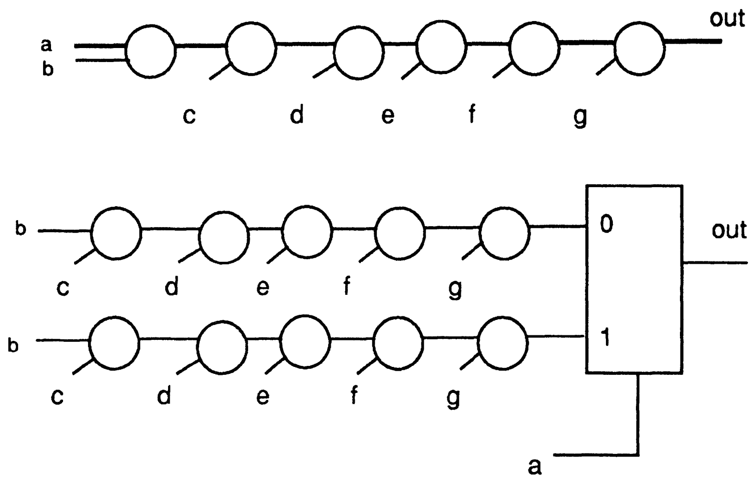


Fig. 4. Generalized select transform

of levels of logic. This approach differs from the others in that no critical paths (or sub-circuits) are identified; all paths are shortened. This is motivated by the fact that in the technology independent phase, delay estimates are poor when compared to the final technology-mapped circuit. However, after technology mapping, various area recovery techniques can be used to undo some of the duplication of speeding up short paths.

In Section 3.2 we show comparisons of three of the delay optimization techniques on a set of benchmark examples.

2.3. TESTABILITY OPTIMIZATION

Testability refers to the ability to determine whether an IC is behaving in accordance with the given specifications. Most approaches to digital IC testing before the middle of the last decade attempted to improve the testability of the design by ad-hoc post-synthesis modifications. However, the increasing need for reliability in manufactured circuits has led to the evolution of testability as an important synthesis criterion.

In order to refer to the ability to test a chip, a fault model is required. Several fault models are in use today. The most common is the stuck-fault model that detects static (or DC) defects [11]. However, since the delay of a chip is often as critical as its logical behavior, circuits should also be tested for dynamic (or AC) defects [43]. This has led to the definition of two delay fault models. Even more comprehensive detection of manufacturing defects may be achieved by checking for faults that model open and shorted connections within transistors. One goal when optimizing combinational circuits for testability is to ensure that 100% of all faults being modeled can be tested by applying a suitable test vector sequence at the primary inputs of the circuits. A defect is detected as a logical difference at some primary output. Other important considerations are the number of test vectors required to detect all the faults, the computation effort required in generating the tests, and the time required to apply them.

There are well developed synthesis techniques for all the fault models listed above. Most are best understood for two-level circuits [3; 22; 17]. Several multilevel optimization operations are known that may be used to retain testability in circuits [22; 17; 35; 12]. Briefly stated, full single stuck-fault testability is achieved via redundancy removal using automatic test generation procedures [37] or via logic minimization using multilevel don't cares [36]. 100% multiple stuck-fault (multi-fault) testability is obtained in two-level circuits by single-output minimization. Algebraic factorization retains multi-fault testability, as well as retaining the test vector set. 100% delay-fault testable circuits can also often be derived for two-level circuits by modifying the logic minimization algorithm; algebraic factorization again preserves testability and test vector sets [17].

3. Relations Between Optimization Criteria

The goal of logic optimization is to obtain a design that is fully optimized with respect to all three criteria; yet, all the optimization techniques mentioned in the previous section target only one criteria. However, when optimizing for one of the goals, sometimes the effects on the remaining criteria are known or may be predicted. In this section we review some well-known results and detail a few recent results.

3.1. Area and testability

Here testability means the percentage of faults for which there exists a test vector which tests for the fault. The area of a circuit is directly related to this. The reason for this is as follows. Consider a connection which can be set to a constant value without affecting the functionality. In such a case, this connection may as well be replaced by the constant value, thus resulting in a smaller circuit. If the connection is retained in the circuit, then a manufacturing defect that appears on the connection cannot be tested. Untestable circuits are undesirable for several reasons which are explored in detail in the next chapter. Besides a non area-minimal implementation, untestability impedes the test generation process and degrades the reliability of a chip. Several optimality criteria can be used to relate area to testability. A first order optimality criterion for area is 100% *single stuck-fault testability* [3]. In such a circuit, no single connection can be removed without changing the function of the circuit. A second order form of optimality is 100% *multiple stuck-fault testability* [22]. In such a circuit no set of connections can be simultaneously removed from a circuit without changing its functionality.

3.2. AREA AND DELAY

Any of the approaches described in Section 2.2 can be used to improve the delay of a circuit, albeit with some penalty in area [41]. An example of this phenomenon is the different implementations of an adder circuit. A ripple-carry adder has the least area among all adders, but is the slowest. A carry-skip adder [25] is faster and is derived using an additional AND gate and MUX. A carry-select adder [47], is similarly derived from a ripple-carry adder with some duplication of logic to achieve a reduction in delay. A circuit with even better delay uses a carry-lookahead structure, [47], but at substantially higher cost in area.

Synthesis techniques also show this phenomenon. An example of area-delay tradeoff is shown in Figure 5, where a 32-bit ripple carry adder is optimized for delay using path reduction techniques [41]. The experiment is performed using a parameter to control the strength of the optimizations used in restructuring the logic. It is instructive to note that during the course of the area-delay tradeoff, the delay of each of the manual adder designs mentioned above is achieved by the automatic synthesis procedure, but with a greater area penalty than the manual designs. This implies that the tradeoff curve obtained by iterating the path-reduction technique is not optimal.

Table I is a comparison of the area-delay tradeoff achieved using three different approaches to timing optimization. Delay is measured by the number of levels of two-input gates. Results using more accurate mapped delays during optimization yield similar results, but a comparison between the three methods is much harder then. The initial circuits are optimized for area using a standard script in SIS [38]. Each timing optimization program is then invoked; the resulting area and delay are shown in the table. The examples are divided into three sets, separated by horizontal lines in the table. The first set of examples are those for which tree-height reduction (THR) yields the best delay result. The generalized bypass transform (GBX) gives the best delay for the second set, and partial collapsing (PC) using clustering performs best on the third set. No single technique for timing optimization obtains the best delay optimization on all examples. Note that while an area-delay tradeoff exists within each technique, as illustrated in Figure 5 for tree-height reduction, an area-delay tradeoff seems to exist between the different techniques. For example, in circuit *C1908*, the initial circuit has a delay of 36.0 and area 894. Using the generalized bypass a circuit with delay 31.0 and area 908 is obtained. Partial collapsing yields a better delay of 28.0 but at an increased total area of 1112.

Name	Initial Delay	Final Delay			Initial Area	Final Area		
		THR	GBX	PC		THR	GBX	PC
5xp1-hdl	16.0	10.0	15.0	15.0	139	210	140	126
5xp1	29.0	17.0	17.0	22.0	309	312	345	265
alupla	18.0	12.0	14.0	19.0	288	355	301	342
f51m-hdl	17.0	10.0	16.0	13.0	133	186	134	119
f51m	34.0	21.0	23.0	31.0	313	350	347	289
misex1	13.0	10.0	12.0	12.0	114	121	120	99
misex2	13.0	7.0	10.0	8.0	182	223	183	208
rd53	12.0	9.0	10.0	11.0	80	73	80	69
rd84-hdl	20.0	14.0	19.0	15.0	184	194	187	128
seq	26.0	19.0	20.0	20.0	1842	1890	1875	1869
sao2-hdl	45.0	23.0	35.0	32.0	462	299	483	396
C1908	36.0	24.0	31.0	28.0	894	1500	908	1112
C2670	33.0	19.0	30.0	32.0	1688	1594	1692	2166
C3540	46.0	38.0	40.0	38.0	4178	3056	4214	3396
C880	38.0	20.0	34.0	32.0	836	1006	849	966
C5315	40.0	32.0	35.0	34.0	3280	3360	3494	4221
C6288	130.0	77.0	129.0	88.0	7374	7596	7376	5369
C7552	53.0	31.0	44.0	58.0	4394	4326	4446	4740
9sym	12.0	13.0	12.0	13.0	548	515	548	444
9symml	18.0	15.0	13.0	16.0	532	491	549	468
bw	30.0	18.0	16.0	18.0	312	344	414	314
duke2	20.0	17.0	14.0	17.0	946	946	1034	848
misex3c	44.0	34.0	26.0	44.0	1090	1044	1165	1097
rd84	15.0	15.0	14.0	18.0	268	261	277	235
sao2	30.0	23.0	21.0	23.0	420	378	437	269
9sym-hdl	31.0	19.0	27.0	19.0	260	275	263	152
con1	5.0	5.0	4.0	4.0	42	42	43	34
f2	11.0	8.0	8.0	7.0	54	54	63	45
rd73-hdl	17.0	12.0	16.0	11.0	132	153	133	106
rd73	21.0	12.0	20.0	12.0	196	192	199	164
misex3	35.0	25.0	32.0	24.0	1172	946	1176	822
C432	40.0	36.0	37.0	34.0	591	501	617	498
C499	21.0	20.0	21.0	17.0	796	860	796	844
C1355	21.0	20.0	21.0	17.0	796	860	796	844

THR: Tree-height reduction, GBX: Generalized bypass transform
PC: Partial collapsing
All circuit composed of 2-input gates, all initial and final circuits irredundant
Initial circuits optimized for area
Delay measured using number of levels of logic
Area measured by number of literals (= 2 × # of 2-input gates)

TABLE I
Performance optimization using three techniques

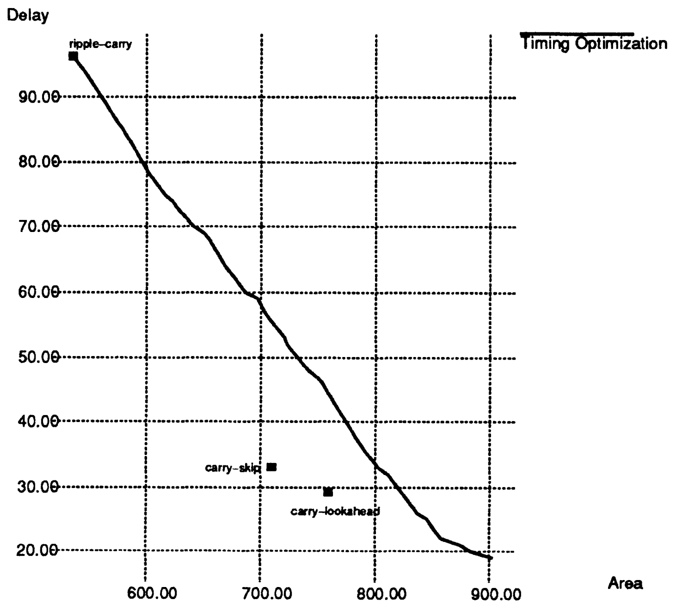


Fig. 5. Area versus delay tradeoff on a 32-bit adder

Tree-height reduction yields the best delay of 24.0, but at a substantially higher area cost of 1500. However, a similar tradeoff does not always exist across all the examples. This is because tree-height reduction and partial collapsing include Boolean operation that sometimes lead to circuit with both smaller delay and area. The generalized bypass transform uniformly increases the circuit area whenever a delay improvement is possible.

Figures 6 and 7 illustrate the relative effectiveness of the three procedures under two different cost functions that consider both area and delay. All delays and areas have been normalized in deriving the cost function for each of the examples shown in Table I. The graph in figure 6 shows the behavior when delay reduction is given a weight of three times more than area increase. In this case, tree-height reduction performs better than the other two approaches, since its cost function is smaller in most cases. However, as seen in Figure 7, partial collapsing is consistently better than tree-height reduction and generalized bypass when the area increase is weighted three times more than the delay decrease. The generalized bypass

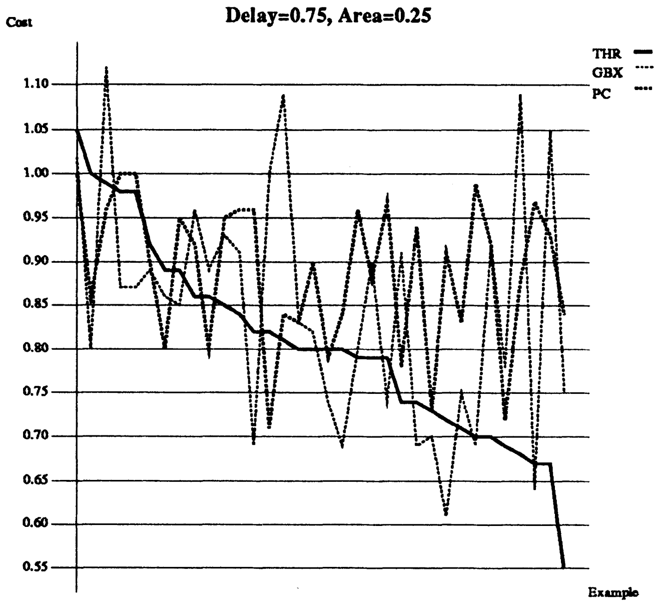


Fig. 6. Cost (= 0.25 * Area + 0.75 * Delay) of timing optimization. Final area and delay are normalized against initial area and delay, respectively. Examples sorted by decreasing cost for tree-height reduction.

transform does not perform consistently better than the other two procedures for any combination of weighted delay and area. This is probably due to the lack of an area recovery step similar to those used by the other two procedures, *i.e.* simplification (without increasing the number of levels of logic) and redundancy removal.

The time needed to perform the timing optimizations is not shown in the table, but it may be a factor in evaluating the effectiveness of the different procedures. The generalized bypass transform completes in a few seconds on each example. Partial collapsing is a little slower due to the simplification and redundancy removal steps. Tree-height reduction is significantly slower, since it repeatedly identifies critical sections of the network, evaluates all the common sub-expressions of each node being sped up, and invokes redundancy removal at the end to recover area. This method requires several minutes on the larger examples shown in the table.

These results suggest that a combination of all the techniques should be used possibly in an iterative process, selecting the best area-delay tradeoff incrementally.

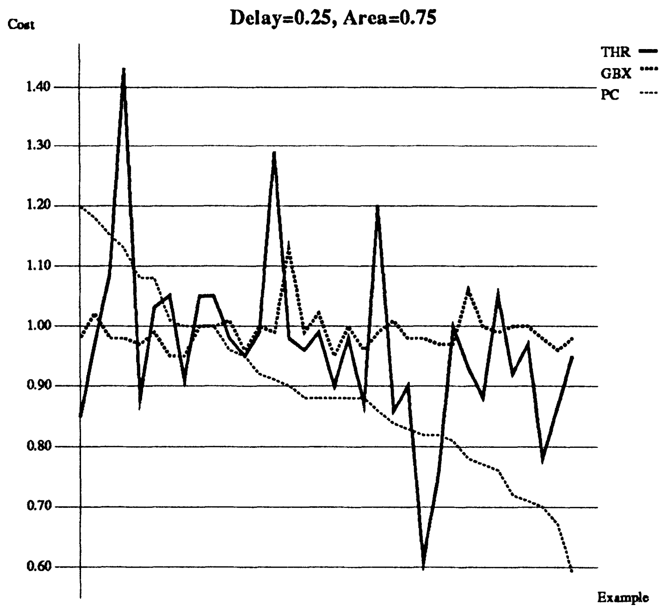


Fig. 7. Cost (= 0.75 * Area + 0.25 * Delay) of timing optimization. Final area and delay are normalized against initial area and delay, respectively. Examples sorted by decreasing cost for partial-collapsing.

This experiment and the associated heuristics should be developed in the future.

3.2.1. Single stuck-fault testability effects

Table II shows a profile of the area and delay of a typical circuit as the tree-height reduction algorithm of [41] progresses, illustrating its ability to trade off area for speed. The first experiment with the example performs timing optimization on an area optimized irredundant circuit. The delay and area on each pass of the algorithm are given in the first two columns. The third column gives the number of redundancies in the circuit. In this example, speedup always led to redundancy. While the number of redundancies typically increases, note that it may also decrease from one iteration to the next. The second experiment (last two columns) performs redundancy removal during each pass of the timing optimization algorithm. While this procedure is computationally expensive, it represents a better area tradeoff. In addition, the final delay of the circuit in the second experiment is 23% less than for the first (21.80 versus 26.80). However, redundancy removal applied to the final circuit of the first experiment yields a delay of 21.80, but at an increase of almost 7% in area. In general, no definite conclusion can be made about the impact of redundancy during timing optimization on the delay or area of the final circuit; it is conceivable that a redundant circuit may allow some operations that lead to smaller delays and areas. However, as shown in Section 3.3, redundancy is undesirable in optimized circuits and every redundant circuit has an irredundant version at least as fast.

Table III shows the final number of redundant faults for various circuits after performing timing optimization on an initially irredundant area optimized circuit. The initial circuits for timing optimization were obtained by using a standard area optimization script in SIS [38], followed by redundancy removal to ensure full testability. Only those circuits in the ISCAS and MCNC benchmark suite that have redundant faults introduced by timing optimization are reported in the table. While most of the circuits have only a few redundancies introduced during the delay optimization process, a few circuits have a large number of redundant faults.

3.3. Performance and Testability

The creation of redundancy during performance optimization naturally leads to the question of the relationship between performance and testability in optimized circuits.

Consider a 2-bit block of a carry-skip adder, shown in Figure 8. In terms of area and performance, the carry-skip adder is between that of a ripple-carry and a carry-lookahead adder. The carry-skip adder uses a conventional ripple-carry adder (the output of gate 11 is the ripple-carry output) with an extra AND gate (gate 10), and MUX added to each block. If all the propagate bits through a block are high (the outputs of gates 1 and 3) then the carry-out of the block ($c2$), is equal to the carry-in to the block ($c0$). Otherwise, it is equal to the output of the ripple-carry adder. The multiplexer thus allows the carry to skip the ripple-carry chain when all the propagate bits are high. A carry-skip adder of n bits can be constructed by cascading a set of individual carry-skip adder blocks, such as Figure 8.

In most cases, the straightforward removal of redundancy does not affect the speed of a circuit. However, in the carry-skip adder, in which an extra carry-chain is added to improve the speed, removing the attendant redundancy in the design (the select input of the MUX is redundant when stuck at *0*) slows the circuit down.

The extra AND gate and MUX of the carry-skip adder have a profound effect on its performance and testability. First consider the impact on the performance and refer to Figure 8. Assume the primary input $c0$ arrives at time $t = 5$ and all the other primary inputs arrive at time $t = 0$. Assign a gate delay of 1 for the AND and OR gates and gate delays of 2 for the XOR and MUX. It can be shown that the worst-case delay of the circuit is along the path from $a0$ to $c2$ through gates

without redundancy removal			with redundancy removal	
Delay	Area	# Red.	Delay	Area
38.80	1229.0	0	38.80	1229.0
37.40	1233.0	7	37.40	1233.0
35.80	1241.0	6	35.60	1227.0
34.60	1261.0	10	34.40	1237.0
33.20	1273.0	10	32.80	1249.0
32.60	1292.0	15	31.60	1269.0
31.40	1318.0	20	30.60	1285.0
31.20	1328.0	23	30.00	1297.0
30.60	1338.0	23	29.80	1313.0
30.20	1350.0	23	28.80	1309.0
29.40	1358.0	26	28.00	1321.0
29.00	1376.0	26	26.80	1327.0
28.80	1394.0	28	26.20	1343.0
28.20	1400.0	33	25.80	1367.0
27.80	1414.0	37	25.80	1369.0
27.40	1439.0	40	25.60	1379.0
27.60	1443.0	41	24.40	1367.0
27.40	1471.0	42	23.80	1373.0
†26.80	1483.0	45	23.60	1395.0
			23.40	1425.0
			22.60	1437.0
			22.00	1467.0
			21.80	1473.0

Example circuit is rot.blif
Each primary input arrival time set at 0.0 delay units
† On performing timing optimization after redundancy removal on this circuit, area of circuit with delay 21.80 is 1577.0

TABLE II
Example: Impact of redundancy on timing optimization

Name	Initial circuit		Final Circuit		
	Delay	Area	Delay	Area	# Red.
C1355	33.20	820.0	30.80	868.0	4
C1908	46.80	849.0	41.80	1079.0	4
C5315	48.20	2904.0	45.40	3037.0	14
5xpl	22.60	188.0	20.00	203.0	3
alu4	27.40	687.0	26.60	691.0	2
apex2	19.20	416.0	18.00	493.0	18
apex4	212.20	4986.0	149.80	5457.0	173
apex7	17.00	431.0	16.00	471.0	4
b9	13.60	266.0	12.40	290.0	6
clip	17.00	201.0	14.60	209.0	2
des	102.40	6109.0	95.40	6146.0	9
duke2	18.60	637.0	18.00	659.0	6
f51m	30.00	238.0	26.20	301.0	13
misex1	20.00	95.0	17.40	146.0	14
misex2	10.60	171.0	9.40	185.0	1
rd73	23.40	111.0	17.80	175.0	4
rd84	19.20	229.0	17.60	353.0	28
rot	38.00	1173.0	27.20	1381.0	40
sao2	21.40	233.0	16.80	276.0	16
z4ml	15.60	76.0	13.60	104.0	5

Each primary input arrival time set at 0.0 delay units
All circuits initially irredundant

TABLE III
Redundancy in timing optimization (tree-height reduction)

1, 6, 7, 9, 11 and the MUX in Figure 8. This is the critical path and its output is available after 8 gate delays [2].

The (statically or topologically) longest path in the circuit is the path from $c0$ to $c2$ through gates 6, 7, 9, 11 and the MUX (available after 11 gate delays). However, it is a false path in the carry-skip adder. In contrast, the topologically longest path determines the delay of the circuit in a ripple-carry adder. Thus the additional circuitry reduces the delay of the circuit. As regards testability, while a ripple-carry

[2] We are concerned with the critical path through the carry-out of the circuit, even though there is a path whose output is available after 9 gate delays for the final sum bit in the block. This is because in an adder composed of blocks similar to Figure 8, the critical path for the entire adder will be the path through the carry-out of each block.

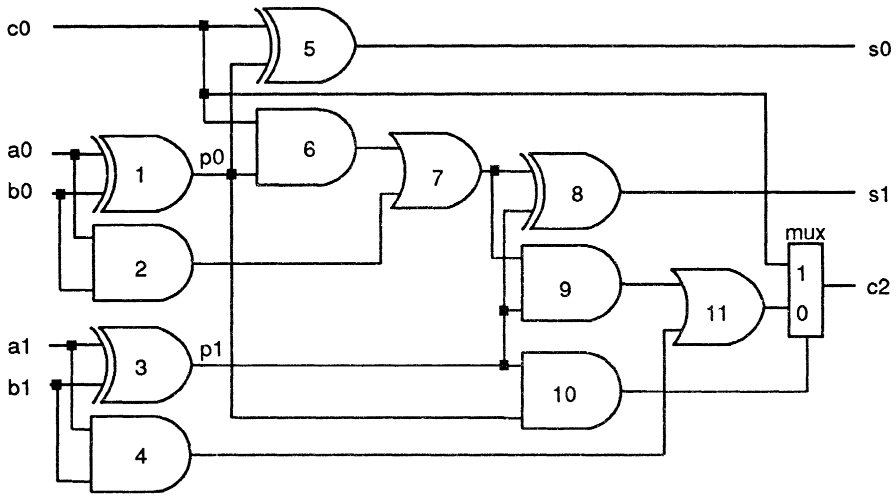


Fig. 8. 2-bit carry-skip adder

adder is fully testable, the carry-skip adder has a single redundancy. In Figure 8, the single stuck-*0* fault on the output of gate 10 is not testable. This is due to the fact that the carry-skip adder becomes a logically-equivalent ripple-carry adder in the presence of the fault. Thus, in attempting to gain speed, the testability of the circuit is compromised.

There is a further problem with the carry-skip adder. Consider the case where the output of gate 10 is stuck at *0*, effectively reducing the circuit to a ripple-carry adder. The critical path is now the longest path in the circuit and its output is available after 11 gate delays. If the clock had been set based on the length of the original critical path (in the absence of faults), then the circuit will behave incorrectly when the single stuck fault exists. This is a serious problem since the stuck-*0* fault on the output of gate 10 is not testable using standard static testing techniques. Thus an untestable manufacturing defect can cause the circuit to malfunction.

The reason the stuck-*0* fault causes the circuit to slow down is that a non-sensitizable (*false*) path becomes sensitizable in the presence of the fault. In Figure 8 the longest path is a false path; however, in the presence of the stuck-*0* fault on the output of gate 10, it becomes a true path. Thus, the output of the circuit is now correctly available only after 11 gate delays rather than 8 gate delays.

4. The KMS Algorithm

As shown in the previous two sections, performance optimizations can, and do in practice, introduce stuck-fault redundancies into designs. The negative aspect of redundancy on reliability in a high-performance circuit was also illustrated. This section is a description of the KMS algorithm which proves that for every high-performance circuit with redundancies, there always exists an equivalent irredundant circuit at least as fast as the original [23; 32]. First, three well-known terms are defined.

DEFINITION 4.1. *A* **non-controlling** *(or* **identity***)* **value** *for a gate f is the value at its input which in not a controlling value for the gate, and is denoted as $I(f)$. For example, $I(f) = 1$ for an* AND *gate.*

DEFINITION 4.2. *Let $P = \{f_0, f_1, ..., f_m\}$ be a path. The inputs of f_i other than f_{i-1} are called* **side-inputs** *of f_i along P and denoted as $S(f_i, P)$. A path that starts at a primary input and ends at a side-input of P is a* **side-path** *of P.*

DEFINITION 4.3. *A path is* **statically sensitizable** *if there exists an input vector which sets all the side-inputs of the path to non-controlling values. The condition for* **static sensitization** *of a path $P = \{f_0, f_1, ..., f_m\}$ composed of simple gates is $\prod_{i=0}^{m} \prod_{g \in S(f_i,P)} (g = I(f_i))$.*

We use the notion of **viability** [28] in determining the conditions under which a path may contribute to the delay of a circuit. Refer to [28; 27; 30; 32] for the formal definition of viability. It is mentioned here that if a path is statically sensitizable then it is viable, though the converse is not true. However, any path that is not viable is a false path.

4.1. ALGORITHM FOR REDUNDANCY REMOVAL WITH NO DELAY INCREASE

Consider a circuit that has some redundancy. What can be said about the change in delay of the circuit when a constant value (*0* or *1*) is asserted on a redundant connection? While an answer cannot be provided for an arbitrary circuit, there are circuit structures for which the effect of the change in delay by a redundancy removal can be predicted. Three such cases are considered. In the discussion to follow the *first edge* of a path refers to the connection between a primary input and the first gate along the path. A longest path refers to a topologically longest path.

Longest path statically sensitizable: If a given circuit has a longest path that is statically sensitizable (hence sensitizable), then redundancy can be removed without any increase in delay. This is obvious since setting any connection to a constant value (*0* or *1*) cannot increase the length of any path in the circuit. Thus, the delay before and after all redundancy removal is the length of the longest path. The next two cases explain how this property is realized on a functionally equivalent implementation of an arbitrary circuit.

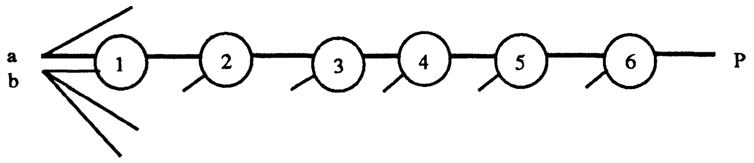


Fig. 9. Fanout-free unsensitizable longest path. Connection from a to 1 is redundant.

Longest path not statically sensitizable - Fanout-free case: Assume that all the longest paths of a circuit are not statically sensitizable (implied by non-viability). Additionally, assume that every gate along any such longest path P has a fanout of exactly one. This implies that a stuck-*0* fault and a stuck-*1* fault on the first edge of P are both redundant. Thus, if the first edge of P is set to a constant value, the logical behavior of the circuit remains unchanged. More importantly, the delay of the resulting circuit also does not increase. This case is illustrated in Figure 9.

Longest path not statically sensitizable - General case: Now consider the case where all the the longest paths are not statically sensitizable and some gates along a longest path P have fanout greater than one. As before, the fault effects of either of the two faults on the first edge do not propagate all the way along P. However, these faults may still be detected through some other path, and thus may be irredundant. Therefore, a constant value cannot be asserted on the first edge of P without changing the functionality of the circuit. But a duplication of some gates can be performed to ensure that all the gates along the longest path have a fanout of exactly one. This is achieved simply by duplicating all gates along P up to the last gate that has multiple fanout.

An example is shown in Figure 10. In the network shown at the top, path P (in bold) is not sensitizable. Thus, both the stuck-*0* or stuck-*1* faults on the first edge of P, which is the connection from a to gate 1, cannot be tested through gate 6 to the output of P. However, each fault may be tested along paths through gates 7 or 8, thus causing the first edge of P to be irredundant. The network shown at the bottom is obtained by duplicating gates 1, 2, and 3, which includes all the gates between the first edge and the last gate with multiple fanout along P. As shown, all gates that have inputs from gates along P in the original circuit, are now connected to the duplicate of the gate. In the example, gate 7 is fed from gate 12, which is the duplicate of gate 1. Similarly, gate 8 has a fanin from gate 3 in the original circuit replaced by a fanin from gate 32 after the transformation.

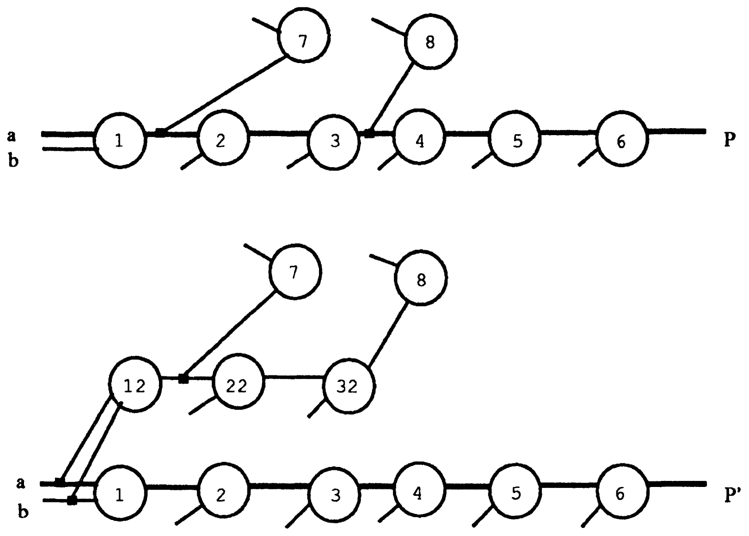


Fig. 10. Duplication to avoid fanout on longest path

This duplication retains the functionality of the circuit. It is also proved that this duplication does not change the viability of any of the paths in the circuit and hence the delay of the circuit remains unchanged [23]. On this new circuit, the longest path again cannot be responsible for determining the delay of the circuit. In fact, both viability and static sensitization of paths remain unchanged by duplication. Hence, the first edge of this new fanout-free path is not testable for either stuck-fault value. It can be set to either constant value without changing the logical functionality of the circuit. It is also shown that this does not increase the delay of the circuit (*c.f.* Theorems 7.1 and 7.2 in [23]). This procedure is then repeated on the resulting circuit.

In summary, the procedure obtains an irredundant implementation of a given redundant circuit by an iterative loop of duplications and redundancy removals which are proven not to increase the delay of the circuit. The original KMS algorithm is presented in pseudo-code in Figure 11.

```
kms($\eta$) {
   /* Circuit $\eta$ has only simple gates. */
   While (no longest path in $\eta$ is
         statically sensitizable / viable) {
      Choose a longest path $P$.
      Find $n$, the gate in $P$ closest to the output
         that has fanout $> 1$.
      If $n$ exists {
         Let $e$ be the fanout edge of $n$ that is in $P$.
         Let $\eta_n$ be the set of gates in $P$ and their fanin
            connections which lie between the primary input
         of $P$ and $e$.
         Duplicate $\eta_n$ to obtain $\eta'_n$.
         Let gate $n'$ in $\eta'_n$ correspond to $n$ in $\eta$.
         Change edge $e$ to be the single fanout of $n'$.
         Call the path in $\eta'$ corresponding to $P$ in $\eta$, $P'$.
      }
      Else {
         $P'$ is the same as $P$.
      }
      Set first edge of $P'$ to either constant $0$ or $1$.
      Propagate constant as far as possible,
         removing useless gates.
   }
   Remove remaining redundancies in any order.
}
```

Fig. 11. KMS algorithm for redundancy removal with no increase in delay

4.2. A SINGLE-PASS ALGORITHM

Due to the large number of false long paths in some circuits, it is imperative that any efficient algorithm must not explicitly enumerate each false path while performing the KMS transformation. This section develops such an algorithm and proves its correctness.

DEFINITION 4.4. *The set of all the paths beginning at connection c and terminating at a primary output is called the* **path-set** *of the connection c, and is denoted* $\mathcal{PS}_c$.

Note that the paths in the path-set of a connection are IO-paths only if the connection is from a primary input.

Consider a connection c from a primary input in a circuit η. Additionally, assume the computed delay of η is $< L$ and let every path in $\mathcal{PS}_c$ be of length $\geq L$. Note that every path in $\mathcal{PS}_c$ is an IO-path. On completion of the KMS algorithm of Figure 11, all the paths in the resulting circuit, η', are of length $< L$. The KMS algorithm removes connections between primary inputs and some gates, *i.e.* only the first edge of any path is removed. Thus, c cannot exist in η', since every IO-path through c is of length $\geq L$. This notion is captured formally by the next definition and theorems.

DEFINITION 4.5. *A* L**-path-disjoint circuit** *is one where the paths in* $\mathcal{PS}_c$*, for any primary input connection* c*, are either all of length* $\geq L$ *or all of length* $< L$.

THEOREM 4.1. *Let* η *be a circuit whose longest viable path is of length* $< L$*. Let* $C = \{c\}$ *be the set of all primary input connections such that each path in* $\mathcal{PS}_c$ *is of length* $\geq L$ *(and hence non-viable). Then any multiple stuck-fault composed of any combination of single stuck-*0 *or stuck-*1 *faults on each* $c \in C$ *is redundant.*

Proof Assume that some multiple stuck-fault F_C on C is irredundant. Let a vector v be a test for the fault. Consider the set of IO-paths P_{F_C} that propagate the fault effect to a primary output under vector v. The length of each $Q \in P_{F_C}$ is $\geq L$ by assumption. Pick a path $Q = \{f_0, f_1, ..., f_k\} \in P_{F_C}$ with the property that for each $f_i \in Q$, f_{i-1} under v is the earliest arriving input of f_i that propagates the fault effect. Such a path Q exists since the fault effect is propagated under v along some path in P_{F_C}. Consider each gate f_i along Q in circuit η when vector v is applied. All the side-inputs to f_i that do not propagate the fault effect are at non-controlling values. The remaining side-inputs each propagate the fault effect but each of them is available only at or after the time input f_{i-1} arrives at f_i. Hence these side-inputs are smoothed out (*i.e.* set to the non-controlling value) when considering the viability of the sub-path Q from f_0 up to f_i. Hence the sub-path of Q up to f_i is viable. Since this is true at f_k, Q is viable, contradicting that no path through c is viable. ■

Theorem 4.1 proves that the logical behavior of a circuit that meets the conditions of the theorem remains unchanged by a multi-fault removal. The next theorem proves that the delay, measured using viability analysis, remains unchanged under this kind of redundancy removal.

THEOREM 4.2. *Let* η *be a circuit such that all IO-paths in* η *of length* $\geq L$ *are non-viable. Let* $C = \{c\}$ *be the set of all connections from a primary input such that each path in* $\mathcal{PS}_c$ *is of length* $\geq L$ *(and hence non-viable). Let* η' *be the circuit resulting after asserting a multiple stuck-fault composed of any combination of the single stuck-*0 *or stuck-*1 *faults on each* $c \in C$*. For any viable path* π' *in* η'*, the corresponding path* π *is viable in* η.

Proof Similar to the proof of Theorem 7.2 of [23]. The difference is that each gate may have several late-arriving side-inputs that are set to constant non-controlling values by the redundancy removal. However, these side-inputs always get smoothed out (*i.e.* set to the non-controlling value) and do not change the viability conditions of any path π' in η' from the viability of the corresponding path π in η. ■

With Theorems 4.1 and 4.2 it is apparent that a multi-fault redundancy removal of the type specified can be done without increasing the computed delay (the length of the longest viable path) or changing the logical behavior of the circuit. Moreover, no duplication is performed. Of course, not all circuits have such connections. A procedure is now described that transforms every circuit to a functionally equivalent L-path-disjoint circuit. In this case, all paths of length $\geq L$ will be removed by the application of the above theorems. The only operation used is the duplication of gates and the transfer of some of the fanouts of a gate to its duplicate.

DEFINITION 4.6. *The set of distinct path lengths from primary inputs to a gate f is denoted by* **atimes(f)**. *The distinct path lengths from each gate f to the primary outputs is denoted by* **etimes(f)**.

If f is a primary input, *atimes(f)* is the single arrival time specified for f. If f is a primary output, *etimes(f)* is *0* . However, the algorithm could be generalized to take account of required times at the output; just take the maximum required time, R_{max}, and think of a buffer on each output f with delay $R_{max} - R_f$, where R_f is the required time for f.

The algorithm to derive an L-path-disjoint network for a given network η is described in Figures 12 through 15. The main procedure, shown in Figure 12, consists of three phases.

The first phase computes the distinct paths lengths from primary inputs to each gate f, and the distinct path lengths from each gate f to the primary outputs. This is done by simply performing a topological traversal from the inputs to the outputs for computing *atimes*, and a reverse topological traversal for computing *etimes* (Figure 13).

The second phase consists of gate duplication and the transfer of some fanouts of a gate f to one or more duplicates of f. The procedure is shown in Figure 14. The essential operation performed on a gate with multiple fanout is the transfer of a set of fanout connections of the gate to a duplicate gate. The gates are traversed in reverse topological order from primary outputs to primary inputs. Let f be a gate that is to be processed by the algorithm. Let P_f be any path from a primary input up to f. A duplication is only performed if there are at least two paths, $Q1_f$ and $Q2_f$ from f to the primary outputs such that[3] $|P_f| + |Q1_f| < L$ and $|P_f| + |Q2_f| \geq L$. Following the duplication, one or more fanout connections are transferred from f to its duplicate f_{dup}, according to the following rule. Let Q_f represent any path from f to the primary outputs after some fanout connections are transferred to f_{dup}. Then $|P_f| + |Q_f| \geq L$ for any P_f and Q_f. Lemma 4.1 below ensures that this condition can always be satisfied. It is important to note that *etimes(f)* is updated

[3] $|P|$ denotes the length of path P.

```
/* Derive an irredundant network no slower than $\eta$.
$\eta$ has no sensitizable paths of length $\geq L$. */
single_pass_kms($\eta$, $L$) {
    /* Circuit $\eta$ has only simple gates.
    $A_f$ and $R_f$ is the arrival time and required time,
        respectively, at gate $f$.
    atimes($f$) are the different path lengths from
        primary inputs to $f$.
    etimes($f$) are the different path lengths from $f$ to
        primary outputs. */

    /* Compute atimes($f$) and etimes($f$) for each node $f$. */
    kms_setup_times($\eta$, atimes, etimes).

    /* Duplicate gates */
    node_list = list of gates in $\eta$ in reverse topological order.
    Foreach node $f$ in node_list {
        kms_duplicate_gate($f$, $\eta$, $L$, atimes, etimes).
    }

    /* Set constants on first edge of paths of length $\geq L$. */
    kms_set_constant($\eta$, $L$, atimes, etimes).
    Propagate constants as far as possible.
    Remove remaining redundancy in any order.
}
```

Fig. 12. Single-pass algorithm for redundancy removal with no increase in delay

whenever a fanout connection is moved to f_{dup}. Similarly, *etimes(*f_{dup}*)* is created to reflect the paths through the fanout connections transferred from f. Duplication is not done on any node with only one path to any output. Hence primary outputs are not duplicated.

This transformation is repeated on f_{dup} (the assignment $g = g'$ in Figure 14) until no further duplication and transfer of fanout connections is required. It is now shown that the resulting network after all the gates are processed by the algorithm is a L-path-disjoint network.

DEFINITION 4.7. *The* **level** *of a node is the maximum number of nodes along any path from the node to the primary outputs. The level of a primary output node with*

```
kms_setup_times($\eta$, $atimes$, $etimes$) {
    Perform a delay trace on the network.
    Foreach node $f$ of $\eta$ {
        $atimes(f) = \{\}$.
        $etimes(f) = \{\}$.
    }
    /* For each node $f$, compute $atimes(f)$ */
    $node\_list$ = list of gates in $\eta$ in topological order.
    Foreach node $f$ in $node\_list$ {
        If $f$ is a primary input {
            $atimes(f)$ = $A_f$.
        }
        Else Foreach fanin $g$ of $f$ {
            $atimes(f) = \{u + d(f,g) | u \in atimes(g)\}$.
        }
    }
    /* For each node $f$, compute $etimes(f)$ */
    $node\_list$ = list of gates in $\eta$ in reverse topological order.
    Foreach node $f$ in $node\_list$ {
        If $f$ is a primary output {
            $etimes(f)$ = 0.
        }
        Else Foreach fanout $g$ of $f$ {
            $etimes(f) = \{u + d(f,g) | u \in etimes(g)\}$.
        }
    }
}
```

Fig. 13. Path length calculations in the single-pass algorithm

```
kms_duplicate_gate($f$, $\eta$, $L$, $atimes$, $etimes$) {
   $g = f$.
   Foreach time $t \in atimes(f)$ in ascending order {
      If ($t$ + min($etimes(g)$) $< L$ && $t$ + max($etimes(g)$) $\geq L$) {
         /* some fanout of gate must be split */
         $g' =$ duplicate_gate($g$).
         $atimes(g') = atimes(g)$.
         $etimes(g') = etimes(g) - \{t_e | t_e \in etimes(g), t + t_e \geq L\}$.
         $etimes(g) = etimes(g) - \{t_e | t_e \in etimes(g), t + t_e < L\}$.
         Foreach fanout $h$ of $g$ {
            If ($t$ + min($etimes(h)$) + $d(g,h)$ $< L$) {
               Replace connection from $g$ to $h$ by $g'$ to $h$.
            }
         }
         /* repeat on duplicate gate of $g$ */
         $g = g'$.
      }
   }
}
```

Fig. 14. Gate duplication in the single-pass algorithm

no fanout is 0 .

LEMMA 4.1. *Consider gate f, in network η, that is processed by the algorithm of Figure 14. Assume that the lists* atimes(f) *and* etimes(f) *are computed using the procedure of Figure 13. Let f' refer to gate f or any of its duplicates*[4]. *Then the following invariant is true for each f': for each $f'_{atime} \in$* atimes(f'), *f'_{atime} +* etimes(f') $< L$, *or f'_{atime} +* etimes(f') $\geq L$[5].

Proof Note that the gates of network η are processed in reverse topological order. The proof is by induction on the level of a node f. Also note that the invariant holds trivially if *etimes*(f) has only one element.

[4] There is no loss in generality in referring to the gates by a single representative f', since, for each path from the primary inputs to f, there exists a corresponding path to some duplicate gate of f.

[5] The notation $x + S < L$, for x a scalar and S a set, means that $x + s_i < L$ for all $s_i \in S$. $x + S \geq L$ has an analogous meaning.

```
kms_set_constant ($\eta$, $L$, $atimes$, $etimes$) {
    /* $\eta$ is a $L$-path-disjoint network */
    Foreach primary input $f$ of $\eta$ {
        /* $atimes$(f) has exactly one entry */
        $t = atimes(f)$.
        Foreach fanout $g$ of $f$ {
            If ($t + d(f, g)$ + min($etimes(g)$) $\geq L$) {
                Replace $g$ by constant $0$ or $1$.
            }
        }
    }
}
```

Fig. 15. Setting constants on false paths in the single-pass algorithm

Induction Basis: If level $= 0$, then f is a primary output with no fanout. Thus, $etimes(f) = \{0\}$. Let P_f be any path from a primary input to f. Since $|P_f| + etimes(f)$ is either $< L$ or $\geq L$, the invariant holds.
Induction Hypothesis: Assume the invariant is true for all gates of level $< k$.
Induction Step: Let f be a gate of level k.

$\mathtt{min}(etimes(f))$ and $\mathtt{max}(etimes(f))$ represent the minimum and maximum times, respectively, in the list $etimes(f)$.

Case 1: f has single fanout.

Let h be the single fanout of f. By definition, h has level $< k$. By the induction hypothesis, either $h_{atime} + etimes(h) < L$, or, $h_{atime} + etimes(h) \geq L$, for each $h_{atime} \in atimes(h)$. But, $atimes(h) \supseteq atimes(f) + d(f, h)$. Rewriting the invariant for h, $f_{atime} + d(f, h) + etimes(h) < L$, or, $f_{atime} + d(f, h) + etimes(h) \geq L$. But, $etimes(f) = etimes(h) + d(f, h)$, since f has single fanout. On further rewriting of the invariant for h, $f_{atime} + etimes(f) < L$, or, $f_{atime} + etimes(f) \geq L$. Therefore, the invariant holds for f also.

Case 2: f has multiple fanout and $t + \mathtt{max}(etimes(f)) < L, \forall t \in atimes(f)$. In this case, each path through gate f is of length $< L$ and the invariant holds.

Case 3: f has multiple fanout and $t + \mathtt{min}(etimes(f)) \geq L, \forall t \in atimes(f)$. In this case, each path through gate f is of length $\geq L$ and the invariant holds.

Case 4: f has multiple fanout and there exists some $t \in atimes(f)$, such that $t + \mathtt{min}(etimes(f)) < L$ and $t + \mathtt{max}(etimes(f)) \geq L$. This is the case where a duplication and transfer of some fanout connections is performed. Let t_{min} represent the smallest time $t \in \mathit{atimes}(f)$ for which the condition for case 4 holds. Let f_{dup} be a duplicate of f. Each fanout h of f satisfies either $t_{min} + d(f,h) + \mathtt{min}(\mathit{etimes}(h)) < L$ or $t_{min} + d(f,h) + \mathtt{min}(\mathit{etimes}(h)) \geq L$. If the first condition holds the connection from f to h is replaced by the connection from f_{dup} to h. Nothing is done if the second condition is true.

Consider the gate f after all its original fanout connections are processed. For each fanout connection h retained on f, $t_{min} + d(f,h) + \mathtt{min}(\mathit{etimes}(h)) \geq L$. For any $t_f \in \mathit{atimes}(f)$, where $t_f > t_{min}$, obviously, $t_f + d(f,h) + \mathit{etimes}(h) \geq L$. Rewriting, $t_f + \mathit{etimes}(f) \geq L$. For any $t_f \in \mathit{atimes}(f)$, where $t_f < t_{min}$, then $t_f + \mathtt{max}(\mathit{etimes}(f)) < L$, due to the choice of t_{min} assumed above. Thus, $t_f + \mathit{etimes}(f) < L$. Thus, f satisfies the invariance condition. The algorithm repeats on f_{dup} which ensures that the invariant eventually holds on each duplicate of f created. ■

Since the invariant stated in Lemma 4.1 holds for each primary input, the final network is an L-path-disjoint network. Thus on completion of the second phase, the path-set of each first edge of any path of length $\geq L$ contains only paths of length $\geq L$. Hence, all such edges are set to constant *0* or *1* in the final phase of the single-pass algorithm (Figure 15).

Figure 16 illustrates the working of the algorithm on an example network. Each gate has unit delay and all primary inputs arrive at $t = 0$. Assume that the initial network (top of the figure) has a longest viable path of length 3. Hence an L-path-disjoint network for $L = 4$ is required. Following the first phase of the algorithm, the variables *atimes* and *etimes* are as follows:

$$\begin{array}{ll}
\mathit{atimes}(g1) = \{1\} & \mathit{etimes}(g1) = \{2,3,4\} \\
\mathit{atimes}(g2) = \{1,2\} & \mathit{etimes}(g2) = \{2,3\} \\
\mathit{atimes}(g3) = \{1,2,3\} & \mathit{etimes}(g3) = \{1,2\} \\
\mathit{atimes}(g4) = \{1,2,3,4\} & \mathit{etimes}(g4) = \{1\} \\
\mathit{atimes}(g5) = \{1,2,3,4,5\} & \mathit{etimes}(g5) = \{0\}
\end{array}$$

The L-path-disjoint network is obtained by a reverse topological traversal. Since both $g5$ and $g4$ have single fanout, no duplication is done on these gates. $g3$ is duplicated to obtain gates $g31$ and $g32$ (second network from the top of the figure). $g31$ is connected to $g4$ while $g32$ is connected to $g5$. Now, $\mathit{etimes}(g31) = \{2\}$, and $\mathit{etimes}(g32) = \{1\}$, and the invariant of Lemma 4.1 is now satisfied for $g31$ and $g32$. Similarly, on duplicating $g2$ and $g1$ as shown, an L-path-disjoint network for $L = 4$ is obtained (bottom of the figure). Any multiple stuck-fault on the inputs a and b of $g11$, and c of gate $g21$ can now be removed. By the theorems discussed earlier in this section, the computed delay and logical behavior of the resulting circuit remain unchanged.

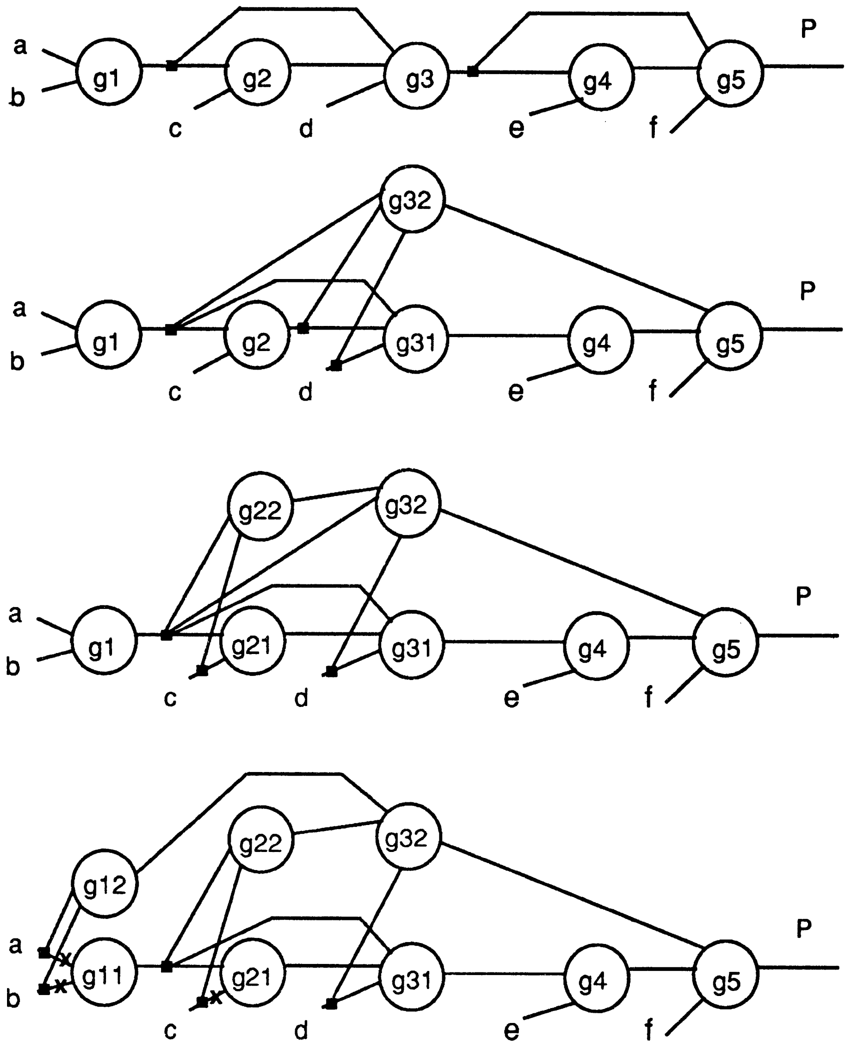


Fig. 16. Example: Construction of an L-path-disjoint network

4.3. Results using the single-pass algorithm

Results of an implementation of the single-pass algorithm for redundancy removal, guaranteeing the delay does not increase are shown in Tables IV and V. For each circuit the length of the longest sensitizable path, T, is first determined using an efficient timing analysis algorithm [30]. The smallest distinct path length $L > T$ is also known. Using the transformation of Figure 14, an L-path-disjoint circuit is derived from the original circuit. Finally, using Theorem 4.1 and 4.2, the first edge of every path of length $\geq L$ is set to a constant. The topologically longest path in the resulting circuit is now of length $< L$ ($\leq T$). Finally, standard redundancy removal is used to derive an irredundant circuit that is no slower than the original circuit. The algorithm thus provides the same effect as the KMS algorithm in a single-pass following the timing analysis phase. The number of operations performed by the algorithm is linear in the number of connections of the original circuit and the number of distinct path lengths $\geq T$ in η.

The single-pass algorithm completes on all the circuits experimented with. The CPU time on a DEC 5000, not including the timing analysis phase and the final redundancy removal, is only a few seconds even for the largest example.

The area increase for the largest adder circuit *csa 64.8* is 22%. While this increase in area is substantial, a smaller penalty is achieved if each 8-bit block of the adder is first made irredundant using the KMS algorithm without an increase in the delay. In this case the penalty is only 15%. However, a decrease in area is also not always observed for the other adder circuits when the KMS algorithm is performed independently on each block of the adder[6].

Table V shows the results of the single-pass algorithm on optimized MCNC and ISCAS circuits. It is observed that there is no area penalty incurred by the algorithm for the two largest circuits, *rot* and *des*. Note that straightforward redundancy removal on the initial *rot* circuit slows it down.

5. Optimizing area, delay, and testability

Using the KMS algorithm any high-performance circuit (obtained manually or by any of the performance optimization techniques of Section 2.2) can be made fully testable (with respect to single stuck-fault or multiple stuck-fault testability). Although a tight area bound by the KMS algorithm is still open[7], empirically, the area penalty incurred by the KMS algorithm is very small. In summary, application of the KMS algorithm removes long false paths while guaranteeing that the delay

[6] In performing the experiment, the arrival time of the carry-input to each block is set higher than the arrival times of the other inputs. This correctly captures the actual critical paths through the block when it is cascaded with the other blocks to form the complete adder.

[7] A weak bound of nG for the area increase has been proved where G is the number of gates in the circuit and n is the number of distinct path lengths in the circuit between the longest viable path and the longest path length.

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	sKMS	Init.	RR	sKMS
ripple.16	0	31.0	31.0	31.0	31.0	139	139	139
lookahead.16	0	16.0	16.0	16.0	16.0	172	172	172
ripple.32	0	63.0	63.0	63.0	63.0	283	283	283
lookahead.32	0	28.0	28.0	28.0	28.0	361	361	361
csa 2.2	2	8.0	8.0	6.0	6.0	22	18	18
csa 2.2†	2	11.0	9.0	9.0	6.0	22	18	21
csa 4.2	4	14.0	12.0	10.0	10.0	44	36	43
csa 4.4	2	12.0	12.0	10.0	10.0	40	36	36
csa 8.4	4	22.0	20.0	18.0	18.0	80	72	79
csa 16.8	4	38.0	36.0	34.0	34.0	152	144	159
csa 8.2	8	26.0	16.0	18.0	14.0	88	72	83
csa 16.2	16	50.0	24.0	34.0	22.0	176	144	179
csa 16.4	8	42.0	24.0	34.0	22.0	160	144	167
csa 32.4	16	82.0	32.0	66.0	30.0	320	288	353
csa 32.8	8	74.0	40.0	66.0	38.0	304	288	335
csa 64.8	16	146.0	48.0	130.0	46.0	608	576	711
csa 8.2‡	8	26.0	16.0	18.0	13.0	88	72	84
csa 16.2‡	16	50.0	24.0	34.0	21.0	176	144	168
csa 16.4‡	8	42.0	24.0	34.0	21.0	160	144	172
csa 32.4‡	16	82.0	32.0	66.0	29.0	320	288	344
csa 32.8‡	8	74.0	40.0	66.0	37.0	304	288	348
csa 64.8‡	16	146.0	48.0	130.0	45.0	608	576	696

RR: redundancy removal
sKMS : single pass KMS algorithm
All primary input arrival time set at 0.0 delay units
†: Carry-input arrival time set at 5.0 delay units
‡: Single pass KMS algorithm performed independently on each block
Final circuit verified against initial circuit

TABLE IV
Single-pass algorithm versus redundancy removal on adders

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	sKMS	Init.	RR	sKMS
5xpl	1	11.0	9.0	9.0	9.0	58	58	61
rot	37	19.0	17.0	19.0	17.0	437	424	423
des	17	15.0	13.0	13.0	13.0	2007	2000	1992
clip	4	8.0	8.0	7.0	7.0	64	61	61
duke2	2	9.0	9.0	9.0	9.0	190	190	190
f51m	39	18.0	17.0	16.0	16.0	173	139	138
misex2	6	7.0	7.0	7.0	7.0	94	90	90
rd73	10	11.0	11.0	11.0	11.0	94	83	83
sao2	9	12.0	12.0	12.0	12.0	126	119	119
z4ml	3	10.0	10.0	10.0	10.0	47	42	42
misex1	0	9.0	7.0	7.0	7.0	28	28	31
bw	0	20.0	14.0	14.0	14.0	85	85	99
z4ml	0	7.0	7.0	7.0	7.0	30	30	30
C1908	3	23.0	21.0	23.0	21.0	325	322	343
C6288	2	120.0	119.0	120.0	119.0	2353	2350	2363
s641	0	19.0	18.0	18.0	18.0	122	122	126
s713	35	27.0	24.0	19.0	18.0	148	122	126
s1238	67	20.0	19.0	17.0	17.0	429	392	396
s9234	224	29.0	28.0	24.0	24.0	1735	1526	1526
s15850	310	40.0	39.0	37.0	37.0	3268	2954	3013

RR: redundancy removal
sKMS : single-pass KMS algorithm
All primary input arrival time set at 0.0 delay units
Final circuit verified against initial circuit

TABLE V
Single-pass algorithm versus redundancy removal on MCNC and ISCAS circuits

does not increase, achieves full testability, and increases the reliability of the circuit at a small expense in area.

Our final interest is in the relationship between all three parameters of logic optimization. The tradeoff that exists between these three parameters is illustrated via the Pareto surface shown in figure 17. The Pareto surface represents those points corresponding to circuits for which no single parameter may be further improved without reducing the optimality of any of the other parameters. In particular, this tradeoff is illustrated for the KMS algorithm. As shown, the KMS algorithm retains the delay of the circuit while improving its testability to 100%. However, this

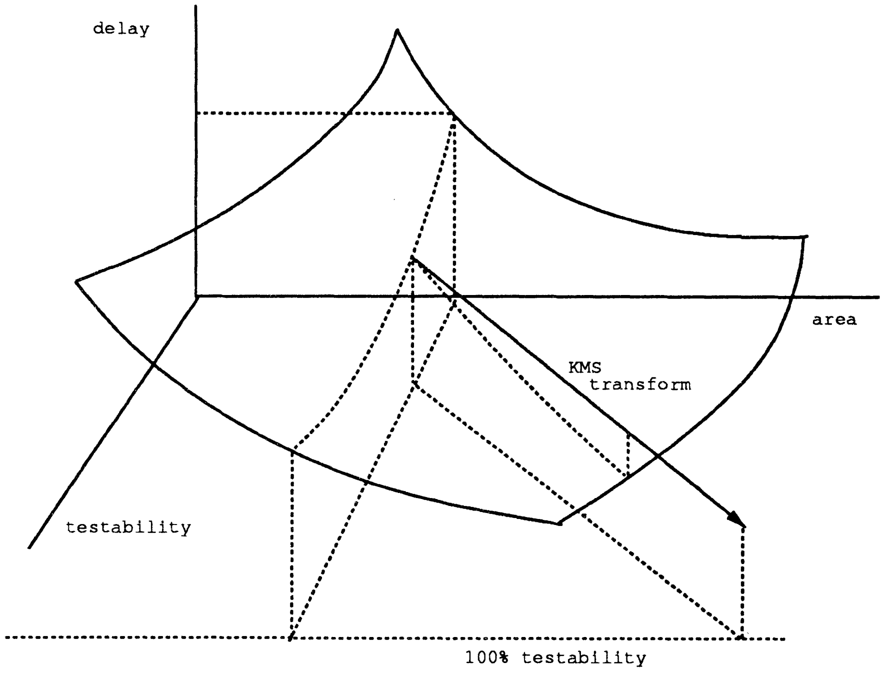


Fig. 17. The Pareto surface for logic optimization

transformation is achieved at an increase in the area of the circuit.

The goal of understanding the tradeoff surface will be achieved when logic transformations that guarantee the movement from one point on the surface to another are devised. The next step is to develop transformations that optimally recover area without decreasing speed or testability.

6. Delay-fault testing: Some open questions

Since high-performance testable circuits are of interest, it is imperative that the circuits be testable for delay faults. A delay-fault is said to occur when a propagation delay falls outside the specified limits. Two models for delay-faults have been proposed to model delay defects on gates or along paths. Of these, the path delay-fault model is considered more comprehensive. In this section we briefly indicate the state-of-the-art with respect to path delay-fault testability and indicate open questions.

An important requirement in using a delay-fault model is that delay faults are detected even in the presence of delay faults in other parts of the circuits; otherwise the test for one delay-fault could be masked by the presence of a second fault. This requirement has led to the definition of robust path delay-fault testabililty. A path is robust delay-fault testable (RPDFT) if there exists a test for the path delay-fault associated with the given path that is valid under arbitrary delays along other paths. The test for a delay-fault along path P consists of a pair of vectors $< v_1, v_2 >$, such that the application of v_1 followed by v_2 causes a transition along P. Additionally, the test must guarantee that the change at the output of P is determined by the changes along P. In short, if a delay-fault exists along P, the output at P changes late under $< v_1, v_2 >$, thus detecting the delay-fault.

The synthesis of robust delay-fault testable circuits was addressed in [17], which describes an approach to realize a 100% RPDFT two-level circuit by modifying one step of a standard two-level logic minimization program [7]. Fully testable multilevel circuits are then obtained using algebraic factorization since the RPDFT test set remains invariant under algebraic factorization. Techniques for creating 100% robust delay-fault testable implementations of common data-path designs, such as adders and parity trees, which cannot be synthesized using two-level logic due to the size of the circuit, are described in [18].

Test generation for robust delay-fault testability has also been addressed. Until recently, the only deterministic and complete approaches to delay-fault test generation published up to now have employed multiple-valued test generation [43; 26]. This suffers from the amount of backtracking possibly required, since each side-input to a path under test may have several valid values assigned that allow for a test. However, a more recent approach [34] relates a robust path delay-fault test to a single stuck-fault test and allows the unaltered application of well-developed stuck-fault test generation algorithms to the problem of robust delay-fault test generation.

Even though there are successful techniques for the synthesis and analysis of robust delay-fault testability of circuits, there are several remaining questions that have to be answered before delay-fault testability becomes viable for most circuit designs. These are enumerated below.

By far the severest problem with robust delay-fault testability is the size of the test set. Since each path in the circuit must be tested for 100% testability, this implies that the test set size is often at least exponential in the circuit size. No successful techniques for collapsing tests, so that several paths are simultaneously tested are known as yet. Another problem is that even though synthesis techniques are known for fully RPDFT circuits, the only approach is to collapse a given circuit to two levels of logic. Even then, circuits exists for which no two-level realization is fully testable. These facts raise the question as to what the area and speed penalties are for achieving high or full delay-fault testability.

The most promising avenue of exploration is the development of a weaker (and more practical) delay-fault model that provides the same delay defect coverage as robust delay-fault testability, but can be easily synthesized, and for which the area and performance penalties can be quantified (and are hopefully small).

7. Conclusions

Logic synthesis has three principal optimization criteria, area, performance, and testability. While optimization techniques for each of these parameters is well developed, until recently not much was understood of the interaction among the three criteria. This paper has presented a survey of the results relating each of these criteria.

The pairwise relationships between speed, size, and testability are well understood and efficient algorithms exist to explore the tradeoffs and interactions that arise between any two parameters. However, results relating all three parameters are still missing although the KMS algorithm demonstrates low area penalties in achieving optimal performance and testability. Since stuck-fault testability is no longer adequate in the testing of high-speed circuits, the tradeoffs between optimal delay-fault testability and performance must be understood better. Additional optimality criteria, such as power dissipation, will also be of interest in the future.

Acknowledgements

This research was funded by grants from Bell Northern, Digital Equipment Corporation, Fujitsu Laboratories Ltd., Intel, AT&T, California Micro, and NSF/DARPA contract MIP-8719546.

References

1. J. Allen. Performance-directed synthesis of VLSI systems. *The Proceedings of the IEEE*, 78(2):336–355, February 1990.
2. K. Bartlett, D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. Bold: The Boulder optimal logic design system. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 62–65, 1987.
3. K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design*, C-7(6):723–740, June 1988.
4. K. Bartlett, W. Cohen, A. deGeus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design*, C-5(4):582–595, October 1986.
5. C. Berman, J. Carter, and K. day. The fanout problem: From theory to practice. In *The Proceedings of the Decennial Caltech VLSI Conference*, pages 69–99, March 1989.
6. L. Berman, D. Hathaway, A. LaPaugh, and L. Trevillyan. Efficient techniques for timing correction. In *The Proceedings of the International Conference on Computer Design*, pages 415–419, August 1990.
7. R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
8. R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel logic synthesis. *The Proceedings of the IEEE*, 78(2):264–300, February 1990.

9. R. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *The Proceedings of the Internation Symposium on Circuits and Systems*, pages 49–54, May 1982.
10. R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, C-6(6):1062–1081, November 1987.
11. M. Breuer and A. Friedman. *Diagnosis and reliable design of digital systems*. Computer Science Press, 1976.
12. M. Bryan, S. Devadas, and K. Keutzer. Testability-preserving circuit transformations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 456–459, November 1990.
13. M. Dagenais, V. Agrawal, and N. Rumin. McBoole: A new procedure for exact logic minimization. *IEEE Transactions on Computers*, C-33(1):229–238, January 1986.
14. J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L.Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.
15. G. DeMicheli. Performance-oriented synthesis of large-scale domino CMOS circuits. *IEEE Transactions on Computer-Aided Design*, C-6(5):751–765, September 1987.
16. E. Detjens, G. Gannot, R. Rudell, and A. Sangiovanni-Vincentelli. Technology mapping in MIS. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 116–119, November 1987.
17. S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay-fault testability of combinational logic circuits. In *The Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 221–238, April 1990.
18. S. Devadas and K. Keutzer. Synthesis and optimization procedures for robustly delay-fault testable combinational logic circuits. In *The Proceedings of the Design Automation Conference*, pages 221–227, June 1990.
19. S. Devadas, K. Keutzer, and S. Malik. Path sensitization conditions and delay computation in combinational logic circuits. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.
20. J. Fishburn and A. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 326–328, 1985.
21. M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.
22. G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison. On properties of algebraic transformations and the multifault testability of multilevel logic. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 422–425, November 1989.
23. K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? *IEEE Transactions on Computer-Aided Design*, C-10(4):427–435, April 1991.
24. K. Keutzer and M. Vancura. Timing optimization in a logic synthesis system. In *The Proceedings of the International Workshop on Logic Synthesis, Amsterdam*. North-Holland, May 1988.
25. M. Lehman and N. Burla. Skip techniques for high-speed carry-propagation in binary arithmetic units. *IRE Transactions on Electronic Computers*, EC-10:691–698, December 1961.
26. C. Lin and S. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer-Aided Design*, C-6(5):694–703, September 1987.
27. P. McGeer. On the interaction of functional and timing behavior of combinational logic circuits. *Ph.D. Thesis, University of California - Berkeley*, November 1989.
28. P. McGeer and R. Brayton. Provably correct critical paths. In *The Proceedings of the Decennial Caltech VLSI Conference*, 1989.
29. P. McGeer, R. Brayton, A. Sangiovanni-Vincentelli, and S. Sahni. Performance enhancement through the generalized bypass transform. In *The Proceedings of the International Conference on Computer-Aided Design*, page 184, November.
30. P. McGeer, A. Saldanha, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Timing analysis and delay-fault test generation using path recursive functions. In *The Proceedings of the International Conference on Computer-Aided Design*, page 180, November.

31. M. Pedram and N. Bhat. Layout driven technology mapping. In *The Proceedings of the Design Automation Conference*, pages 99–105, 1991.
32. A. Saldanha. Performance and testability interactions in logic synthesis. *Ph.D. Thesis, University of California - Berkeley*, October 1991.
33. A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Circuit structure relations to redundancy and delay: The KMS algorithm revisited. In *The Proceedings of the Design Automation Conference*, June 1992.
34. A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Equivalence of robust delay-fault and single stuck-fault test generation. In *The Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1992.
35. A. Saldanha, R. Brayton, A. Sangiovanni-Vincentelli, and K-T. Cheng. Timing optimization with testability considerations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 460–463, November 1990.
36. H. Savoj, R. Brayton, and H. Touati. The use of image computation techniques in extracting local don't cares and network optimization. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.
37. M. Schulz and E. Auth. Advanced automatic test pattern generation and redundancy identification techniques. In *The Proceedings of the International Fault Tolerant Computing Symposium*, pages 30–35, June 1988.
38. E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A sequential synthesis and optimization system. *Submitted for publication*, October 1991.
39. C. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.
40. K. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *The Proceedings of the Design Automation Conference*, pages 357–360, June 1990.
41. K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 282–285, November 1988.
42. J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9:226–231, June 1960.
43. G. Smith. Model for delay faults based upon paths. In *The Proceedings of the International Test Conference*, pages 342–349, August 1985.
44. H. Touati. Performance-oriented technology mapping. *Ph.D. Thesis, University of California - Berkeley*, November 1991.
45. H. Touati, H. Savoj, and R. Brayton. Delay optimization of combinational logic circuits through clustering and partial collapsing. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.
46. J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 510–513, November 1990.
47. N. Weste and K. Eshraghian. *Principles of CMOS VLSI design: A systems perspective*. Addison-Wesley Publishing Company, 1985.

VLSI Architectures for Digital Video Signal Processing

P. PIRSCH
University of Hannover
Appelstr. 9A
D-3000 Hannover 1
Germany

Abstract. VLSI Architectures for real-time implementation of video signal processing algorithms have been studied. These algorithms exhibit very high processing demands in terms of computation rate and access rate which call for architectural structures adapted to the algorithms. Optimization of architectures has to consider the overall silicon area for the computation part, memories and interconnections. VLSI implentations according to the function oriented as well as the software oriented approach will be discussed. As examples for the function oriented approach architectures of dedicated circuits for filtering, DCT and block matching will be evaluated. Software oriented implementations by multiprocessor systems based on MIMD and SIMD architectures will be also presented.

1. Introduction

Real time video signal processing is requested for applications such as image generation, image enhancement and restauration, image analysis, and image coding. The basic algorithms employed for image processing can be divided into

- point type
- filtering type
- matrix algebra type.
- transform type.
- sorter type.

Point type are gray scale transformations, histogram equalization, requantization and intensity mapping. Filtering type are linear phase filters, recursive filters, median filters, adaptive filters, convolution/correlation, window techniques and two-dimensional operators (Prewitt, Kirsch, Hueckel, Sobel). Matrix algebra type are geometric rotation/display, maximum likelihood estimation, pseudo-inverse calculation, stochastic parameter estimation and singular value decomposition. Transform type are Fourier transform, cosine transform, Hough transform, Hadamard transform and so on. The basic algorithms listed above are used for all kinds of image processing applications. Dominating algorithms are filtering and transform techniques.

As an application with a very strong demand for compact and cost effective VLSI realization this paper focuses on VLSI implementations for image sequence coding. Because three-dimensional image sequences are transformed to one-dimensional video signals this will be mostly described as video coding. Video coding has to reduce the transmission rate of the video signals to a specified bit rate without reducing the picture quality below a desired level.

Originally video coding has been investigated for unidirectional services such as TV distribution as well as bidirectional services, e.g. video phone and video conference. Recently new activities are directed to storage of motion video on compact disc (CD). The progress in performance of new workstations makes storage, display, and processing of motion video on workstations feasible. For this reason video signal processing and in particular video coding is at present of large intererest for the community of engineers in the areas of consumer electronics, communications and advanced work stations.

Because of world wide interest in this field international committees as CCITT, CCIR, and ISO are already working on standardization of all kinds of image transmission and recording services [1, 2, 3, 4, 5]. The CCITT and CCIR are considering bit rates of the standardized hierarchy of digital channels. Starting from the basis channel of ISDN (64 kbit/s), also several of these channels are planned for bearer services (HO = 384 kbit/s, H1 = 1536 or 1920 kbit/s). Higher levels in the hierarchy of bearer services with bit rates of approximately 32 Mbit/s (H3) and 140 Mbit/s (H4) are not finally defined.

Source coding methods have to be applied to reach the envisaged bit rates. Because of high picture quality requirements, TV transmission is planned with 32 Mbit/s and HDTV with 140 Mbit/s. In order to have the ability of world wide transmission and mass applications, for video phone and video conference services bit rates in the range of 64 kbit/s to 1920 kbit/s are envisaged. The motion picture expert group (MPEG) of ISO proposes a coding scheme with about 1.2 Mbit/s for video. Latest activities of the MPEG are now directed to higher rates of about 9 Mbit/s for broadcast video and about 45 Mbit/s for HDTV. These bit rates seems to be achievable by future high density recording media. Latest developments on digital channel modulation techniques indicate that these rates could be also reached on analog TV channels of about 7 MHz bandwidth.

From the channel rates discussed above it results that bit rate reduction in the range of 8 up to 600 is needed. Very sophisticated source coding schemes make high reductions possible without degrading the reconstructed images too much [6]. Predictive coding and transform coding is not sufficient. Hybrid coding is frequently proposed. A block diagram of such a hybrid coder is shown in Fig. 1.1. The major characteristics are described below. A previous frame memory is used for interframe prediction. Prediction of moving objects is improved by motion compensation. Because of difficulties in segmenting moving objects and the large overhead for specification of boundary lines of these objects, for most video coding schemes simple block matching algorithms (BMA) are used [7]. The prediction error is coded by a discrete cosine transform (DCT) with adaptive quantization (Q) of the transform coefficients. Run length coding and variable word length coding (VLC) are applied to the quantized DCT coefficients. In order to have the same prediction at receiver and transmitter, a recursive structure is used as in all predictive systems. A two-dimensional loop filter reduces the effects of quantization errors on the prediction value. For non predictable areas the prediction from the previous frame is dropped and intrafame DCT coding is applied.

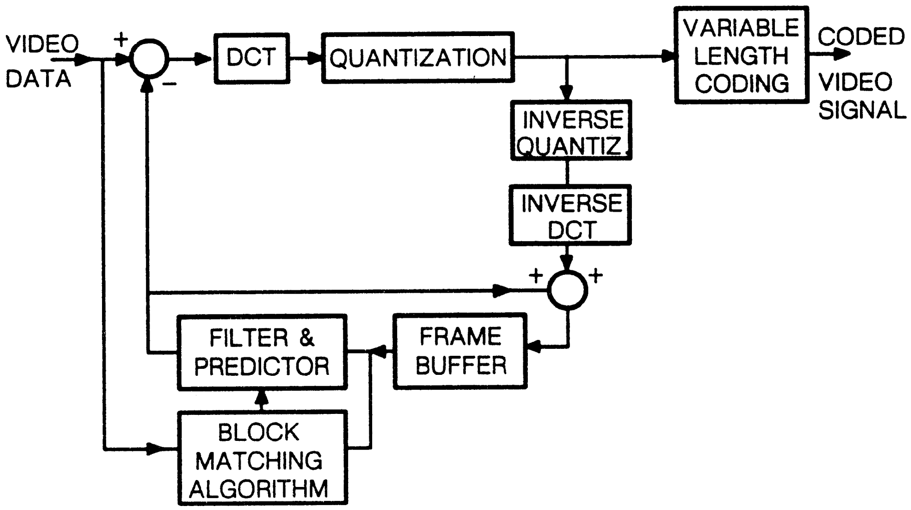


Fig. 1.1. Hybrid coding scheme (transmitter side).

In digital storage media special playback functions are required different from those for communication systems. These functions are random access, high speed search and reverse playback. For implementation of these functions cyclic intraframe coding is proposed [8]. Further motion compensated frame interpolation of dropped frames has to be considered for picture quality improvements.

The evolution of the standard TV results in systems with higher spatial resolution (HDTV). The increased bandwidth of HDTV results also in higher source rate. Subband coding is frequently proposed for bit rate reduction [9]. By two-dimensional filterbanks the original signal is splitted in several bands of smaller bandwidth. The low frequency band could be coded by an hybrid coding scheme according to Fig. 1.1 where the high frequency bands are adaptively quantized and coded by variable length codes (Fig. 1.2)

Essential for the introduction of the discussed video services is low manufacturing cost and compact implementation of the video codec equipment. This calls for VLSI implemention. There is a strong interaction between algorithm and the best suited hardware architecture. Hence, high compaction will be achieved only by adapting the architecture to the algorithm. For this reason application specific VLSI circuits are best suited. Besides dedicated VLSI circuits also programmable multiprocessor systems adapted to the specific application field are of interest. The advantage of programmable systems is the flexibility to accomodate a wide variety of application schemes and to allow modifications of algorithms just by software changes.

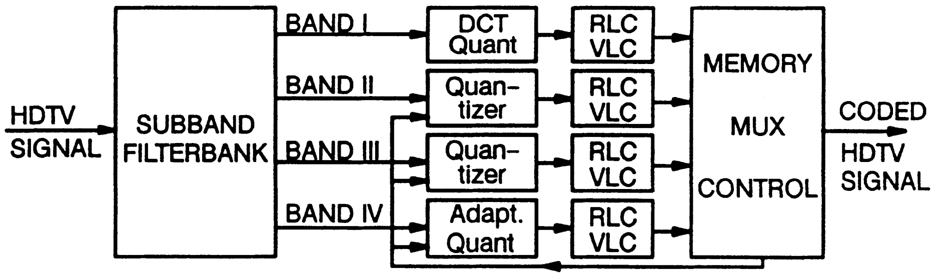


Fig. 1.2. HDTV subband coding scheme (transmitter side).

In the next section characteristics of video signal processing schemes will be explained. Emphasize will be on description of algorithms. After that, strategies for VLSI implementation will be discussed. Alternatives for mapping algorithms onto architectures are incorporated. Then, the two major implemention possibilities will be exemplified. One is direct implementation by circuits dedicated to specific function, the other is programmable multiprocessor systems.

2. Characteristics of video signal processing schemes

The video signal processing schemes will be specified by the sequence of input and output data and the specific computational procedure for achieving the desired input/output relationship. The exact specification of a processing scheme will be also denoted as the algorithm.

Algorithms can be classified according to the complexity of operations and data access which also relates to different kind of implementations. Low level algorithms are simple algorithms which have to be performed for every pel (picture element) in the same manner. A typical example is two-dimensional filtering. The input data are images. The output data can be mostly also interpreted as an image where specific features are emphasized at the output image. The operations are performed in a predefined sequence over the complete image, only access to neighboring pels is needed (locality). Medium level are algorithms which generate features and other characteristic data of the image scene. The derivation of symbols and objects is included in this category. High level operations are object dependent processing and operations with characteristic symbols. High level tasks are rule based intelligent processing steps. The results of the low and medium processing levels are classified and compared to a scene model to verify results.

Sophisticated processing schemes like the hybrid video coding scheme discussed in the previous section cannot be described with a few relations. High complexity processing schemes have to be defined in a hierarchical manner. On the top a processing scheme can be specified by a block diagram as given in Fig. 1.1.

By piecewise refinement further details of data transfers and computations are derived. Depending on the level within the hierarchy, algorithms could be defined on groups of data such as arrays or vectors, single data (word level) or even on the bit level.The computations and data dependencies of algorithms can be described either by recurrence equations, program notations or dependence graphs (DGs). An illustration of a dependence graph is depicted in Fig. 2.1. The nodes represent the operations (computations) and the arcs the data transfers (data dependencies). By assignment of a schedule a specific sequence in time for operations and data is achieved which transfers the DG to a signal flow graph (SFG). The SFG can be used as a structural description of the hardware architecture. The given SFG is a basis of the following hardware design. The mapping of algorithms onto hardware structures ist not unique. A large variety of alternatives have to be considered for implementation.

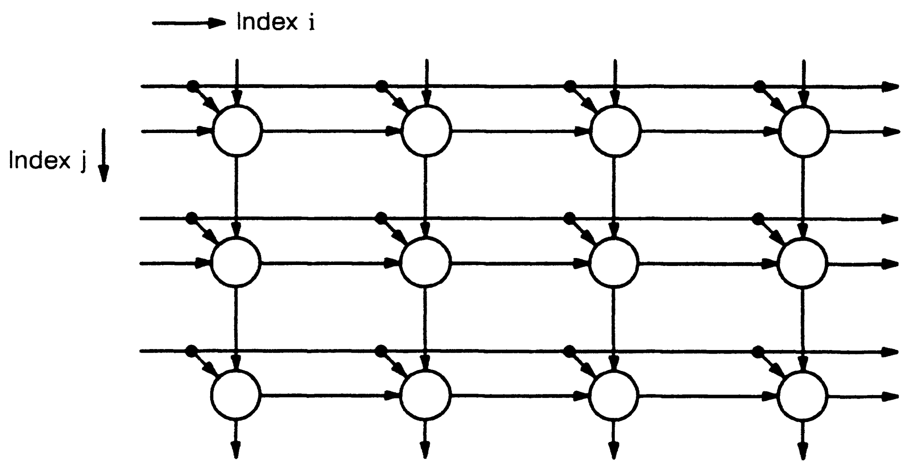


Fig. 2.1. Illustration of a dependence graph in a two-dimensional index space.

Of primary concern are the resources required for implementation of the algorithms. Resources for implementation are logic gates, memory, and communication bandwidth. The requirements on resources are related to measures as

- computation rate
- memory capacity
- data access rate.

The computation rate is proportional to the source rate which is the product of image size and frame rate.

$$R_c = R_s \cdot n_{OP} \tag{2.1}$$

with n_{OP} as mean number of operations per pel, R_s as source rate in pels per second and R_c as computation rate in operations per second. The relation above is obvious for low level algorithms which are performed in the same manner for each pel or a group of pels. For high level algorithms this can be determined as an average over typical image material. In Tab. 2.1 the number of operations per pel is listed for three important low level algorithms. This table shows the fact, that there are several alternatives for implementations of the same algorithm with different computation rates because of different number of operations per pel. The more a priori knowledge about the algorithms is incorporated into the scheme the smaller the number of operations. The computation rate is only one side. The alternatives will display also differences in the requirements for memory, interconnect bandwidth, control, numerical behaviour and achievable parallelism.

TABLE 2.1
Mean number of operations per pel. $N \times N$ window size, p maximal displacement.

Algorithm	Operations per pel
2D DCT	
DOT-product with basis images	$2N^2$
Matrix-vector multiplication	$4N$
Fast DCT	$4 \log N - 2 + 2/N$
Block Matching	
Full search	$3(2p+1)^2$
2D log search	$3 + 24 \log p$
2D FIR Filter	
2D convolution	$2N^2$
Two 1D convolutions (separable Kernel)	$4N$

The storage requirements are influenced by the multiple access to original image data and intermediate results. The interconnect bandwidth depends on the frequency of communication between memory modules and the operative part. The requirements on control depend on the type of access (local or global) and the data dependencies (predefined time driven vs. data dependent processing). For the DCT Tab. 2.2 shows the computation and access rate and the memory capacity under the assumption that the operative part contains just one register for accumulation. The table indicate that fast DCT algorithms are best for the measures compared. In terms of numerical behaviour (truncation of word length) and control the fast algorithms are not the best solution. It should be noted that the data in Tab. 2.1 and 2.2 are not given by asymtotic upper bounds ($O(\cdot)$) because for image processing applications N is limited ($N = 8, 16$).

For realization of special purpose hardware, it is important first to understand the properties and classifications of algorithms before working on architectures

TABLE 2.2
Computational rate, access rate, and memory capacity of various algorithms for implementation of the 2D DCT.
$N \times N$ window size, R_s source rate.

Algorithm	Computational rate	Access rate	Memory capacity
Dot product with basis images	$2N^2R_s$	$(2N^2+1)R_s$	N^4+N^2
Matrix vector multiplication	$4NR_s$	$(4N+2)R_s$	$2N^2+N$
Fast algorithm (Lee)	$(4\log N-2+2/N)R_s$	$(9\log N-6+6/N)R_s$	N^2+2N

and implementation. Measures such as computation rate, access rate and memory capacity are only a rough indicator for the required hardware expense.

3. VLSI Implementation

Essential for the introduction of new video services (communication and storage) is low manufacturing cost and compact realization. For these reasons application specific VLSI components are requested for implementation. Besides system requirements VLSI implementation is influenced by the production volume. High production volume will allow full custom designed components, whereas low and medium volume have to be designed with standard or semi custom components.

Silicon area and chip housing (number of pads) dominate chip manufacturing cost. Both have to be minimized. For minimization of silicon area the trade-off between the area devoted for computation, storage, and interconnections has to be considered. The number of chips can be kept small by large area silicon. This will result in large complexity chips containing a large number of macro blocks.

As the number of blocks per chip increases, the number of interconnections between the blocks also increases. Silicon area for interconnections can be restricted by architectures with local connections between the blocks. Keeping constraints on the I/O bandwidth the number of I/O pins can be limited. The silicon area can be used very efficient by adapting the architectures to the algorithms. Every a priori known feature should be considered for minimization. Dummy cycles reduce the effective computation rate. Therefore processing units should be kept always busy.

The design efforts and design errors will be reduced by regular and modular architectures. Design of memory parts and special function blocks is becoming inexpensive due to their high regularity and modularity. The building block concept simplifies VLSI design by use of high level synthesis tools. New CAD tools give the VLSI designer the required amount of flexibility to cope with the increasing complexity of VLSI components.

Implementation for the hybrid coding scheme (Fig. 1.1) will be taken in the following as a particular example. The hybrid coding scheme is based on the macro block approach. One block of 16 x 16 luminance pels and two blocks of 8 x 8 chrominance pels are processed within the loop for fixed parameters. The block diagram of Fig. 1.1 can be arranged to a functional space as shown in Fig. 3.1. The functional space defines the processing sequence of functions on a high level block basis. All functions will be processed from one macro block of image data to the next. Hence input will be a group of pels according to the macro block size. Mapping of functional blocks orthogonal to the processing sequence (horizontally) results in processors or devices in a function oriented structure. Each processor is dedicated to one or a group of functional blocks. A partitioning according to functions supports individual optimization of processors for given functions. Because of specialized processors compact implementation can be achieved. Since several types of different processors are needed design efforts are very high.

TABLE 3.1
Source rate of several image formats. Net rate without blanking intervals.
Y luminance, C chrominance.

Name	Image Size (active area)	Frame rate in Hz	Source rate in Mpels/s
QCIF	Y: 176×144 C: 88×72	10	0.4
CIF	Y: 352×288 C: 176×144	30	4.6
CCIR 601	Y: 720×576 C: 360×576	25	20.7
HDTV	Y: 1920×1152 C: 960×1152	25	110.6

For most functional blocks of the hybrid coder special integrated circuits (ICs) have been designed and are already available [10, 11]. The large number of different ICs is an appropriate solution for high source rates as needed for broadcast video and HDTV. The source rates for different image sequence formats are listed in Tab. 3.1. The advances in semiconductor technology allow for moderate source rates as for the CIF format the implementation of almost all functional blocks onto on chip. To every functional block only a section of a complete IC is devoted. Hence, a functional block becomes a macro cell. Presently available technologies allow signal processing with up to 100 MHz but the CIF format has a mean sample rate of 4.6 MHz. Hence, the computation intensive blocks can be implemented with reduced parallelism which results in smaller silicon area. Optimization of silicon efficiency for high complexity ICs with macro cell structure requires a global view for minimization of storage and interconnections.

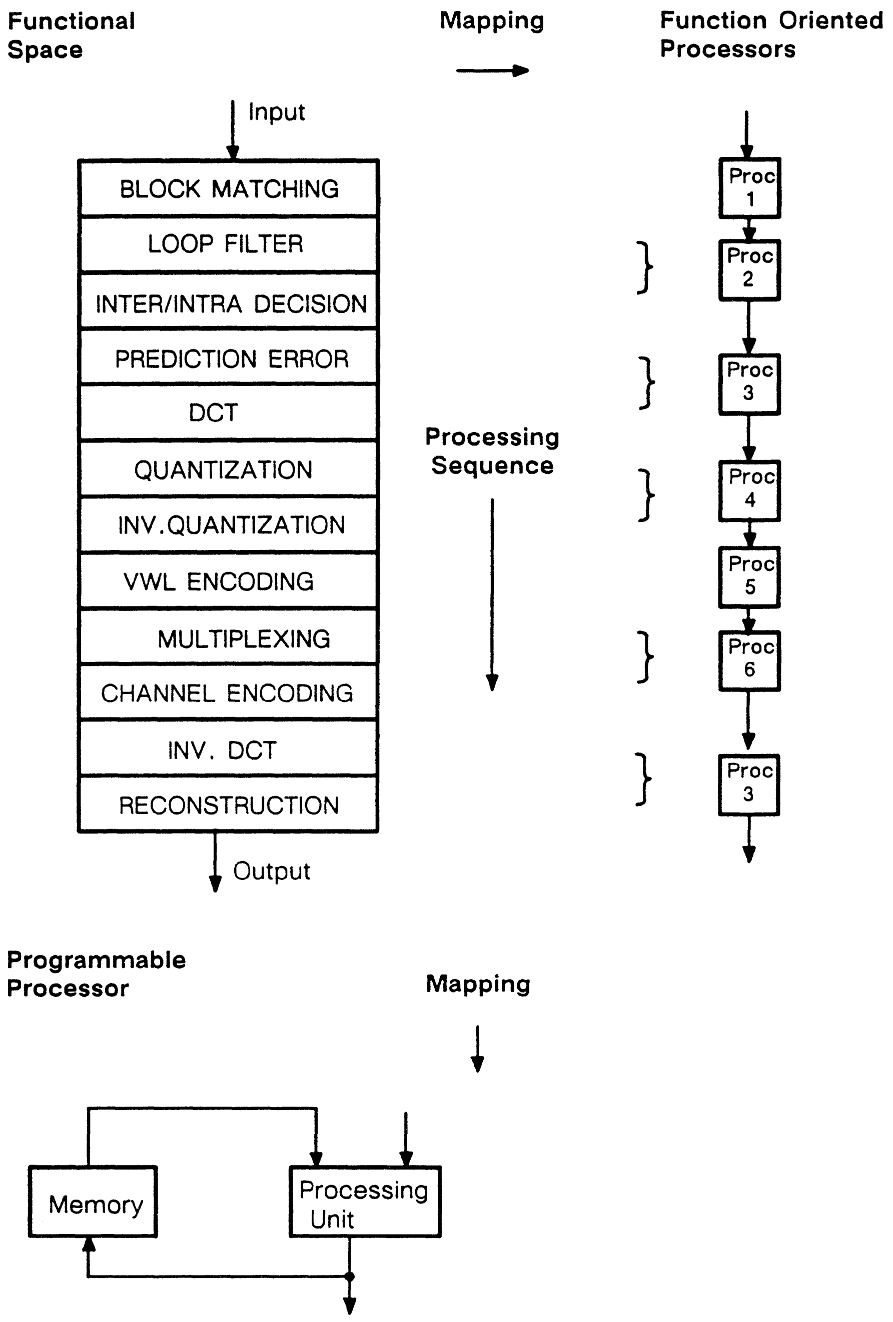


Fig. 3.1. Functional space of the hybrid coder and mapping to function oriented processors and programmable processors.

In order to minimize buffer memories between macro cells the results of a preceding macro cell have to be taken by the following macro cell and immediately processed. The discussion shows that optimization of silicon efficiency for the functional approach requires large efforts for specification and design. In particular, changes of the requirements and algorithms result in new design cycles.

A projection of the functional space in the direction of the processing sequence (vertical) results in one processor performing all functions one after the other (Fig. 3.1). This mapping results in a processor with time dependent processing according to the sequence of functions. The feedback memory is needed for the storage of intermediate results.

The ideal solution for time dependent different operations in the same device is a programmable processor. The mean number of operations per pel of the hybrid coding scheme is about 160. The product with the source rates given in Tab. 3.1 results in computation rates from 60 MOPS for QCIF to 17 500 GOPS for HDTV. Presently avaible DSP chips offer 80 MOPS. Therefore, one processor is not sufficient to provide the required computation rate for image formats above QCIF. Parallel operation of several processors can be easily achieved if the algorithms allow independent processing of image segments. For the hybrid coding scheme according to the H 261 [1] the smallest segment to be independently processed is the macro block of in total 16×16 luminance pels and $2 \times 8 \times 8$ chrominance pels. By assignment of appropriate image segments to each processor the available computational rate can be matched to the requirements. There are several ways to distribute the processors over the image space. In Fig. 3.2 the image segments are used as columns in the image space. Because processing of image sections require for some algorithms also access to pels in neighboring image regions the image segments have to overlap. In case of video coding the overlap results from the search area for the block matching scheme.

As shown in Fig. 3.1 the functional space can be either mapped into a functional approach as sequence of processors or devices which have to perform dedicated functions or a distributed approach where each processor has to perform all functions in a defined sequence for an image segment. Best suited for the functional approach are dedicated devices for the specified functions whereas the distributed approach is best implemented by programmable devices. It is obvious that both approaches also can be combined.

Matching to the requirements of computational rate and access rate can be achieved by an appropriate parallelism. The functional approach provides a sequence of processors which ideal support pipelining. Common for all processors is a predefined data sequence with high throughput rate. The different tasks of the processors have also differences in computational rate which requires individual adaptation to algorithms by specific schemes of parallel processing and pipeling.

The distributed approach offer parallel operations independent of the algorithms. A figure of merit for the needed parallelism can be determined by

$$n_{PE} = R_c \cdot T_{OP} \tag{3.1}$$

with n_{PE} as number of parallel processing elements and T_{OP} as average time for one operation. Because of (2.1) n_{PE} is proportional to the source rate. With present available technologies T_{OP} is in the order of 20ns. Even for a hybrid codec with moderate source rate according the CIF format n_{PE} becomes 15.

In a similar way the required parallelism because of data access rate can be determined. Let the required data access rate R_{DATA} be determined when each operand has to be read out of external memory. Hereby it is assumed that all source data and intermediate results are stored in external memories. Because the memory access time T_{ACC} is limited the required number of parallel access lines would be

$$n_{ACC} = R_{DATA} \cdot T_{ACC} \tag{3.2}$$

The access time of static memory is at present in the order of 30 ns. The data access rate of the hybrid coder for CIF image format is about 690 Mbyte/s. This results in $n_{ACC} = 23$ byte. An implementation with n_{ACC} parallel paths of 8 bit is not worthwhile. Considering that no external access is required for data stored in pipeline registers and in on-chip local memories the access rate to external memories can be essentially reduced. For this reason an appropriate data transfer concept and local memories adapted to the requirements have to be investigated to achieve external data transfer with a few bus lines.

4. Key components for function oriented implementations

A function oriented realization can be very attractive because it has the potential to achieve the smallest size possible by being tailored to the requirements of the application. When mapping the operational parts of a video processing scheme to the blocks of a function oriented realization, the data transfer between blocks has to be taken into account. Considerable hardware expenses for formatting of data sequences might occur if the output from any one block is not matched to the input sequence of the successive blocks. The specific features of the algorithm have influence on the architecture and this will have impact on the overall hardware complexity. Measures such as computation rate, access rate, and memory capacity will give the system designer indications which algorithms are to be prefered from the hardware point of view.

An appropriate optimization criterion is the minimization of the total required silicon area under consideration of I/O constraints. Hereby the trade-off between areas devoted to computation, storage, and interconnections has to be taken into account. In general, there are several alternative architectures which contribute differently to the three mentioned parts. It should be recognized that there is a strong interaction between the algorithms and the most appropriate architecture. In most cases specialized architectures considering all a priori known features of algorithms are the best solution. There are a few exceptions where more general architectures are better because of high regularity which offers cell connections by abutment and reduced efforts for specification and design. Cell connections by abutment reducing considerably the silicon area for connections.

In this section special realizations of three widely used algorithms from the video coding area, namely subband filtering, DCT, and block matching, will be discussed. These algorithms have been selected because of the high requirements they establish. Distinct hardware structures are examined that achieve the high performance requirements in a small silicon area.

4.1. Filter banks for subband coding

In particular for bit rate reduction of HDTV signals subband coding is proposed. Key components for implementation of subband coding schemes are band splitting filter banks at the transmitter side and band synthesis filter banks at the receiver side. At the transmitter side the video signal is separated into several bands by filtering. Sampling in each band is then decimated according to the bandwidth. Then each band is coded according the individual properties and the code words are transmitted to the receiver. After decoding at the receiver the video signal is synthesized by band interpolation filters.

Due to the transition between pass and stop bands, subsampling at the transmitter side will cause aliasing effects. Alias cancellation necessary for good picture quality can be achieved by appropriate choice of the parameters of the coder and decoder filter banks [12,13]. Filter characteristics which incorporate aliasing error cancellation are Quadrature Mirror Filters (QMF) and Conjugate Quadrature Filters (CQF). Both are Finite Impulse Response (FIR) filters with a non-recursive structure. QMFs offer a linear phase behavior. A disadvantage of QMF are small ripples in their overall frequency transfer function. In contrast, CQFs allow a perfect reconstruction of the original signal, if undistorted transmission of the subband signals is provided.

4.1.1. Computation part

Because of the high source rate, processing of HDTV signals results in high computational rates which can be only enabled by intensive pipelining and parallel processing. Therefore, FIR filters are prefered because of simplicity for pipelining intensive realizations. Images are two-dimensional from nature. For this reason 2D filtering is applied. The convolution of the image data $x(\cdot)$ with an impulse response $h(\cdot)$ is given by

$$y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} h(m,n)x(i-m, j-n) \tag{4.1}$$

Using separable filter characteristics the number of multiplications of a 2D filter with $N \times M$ coefficients can be reduced from $N \times M$ to $N + M$. The filtering process is then

$$\begin{aligned} \tilde{y}(i,j) &= \sum_{m=0}^{M-1} h(m) \cdot x(i-m,j) \\ y(i,j) &= \sum_{n=0}^{N-1} h(n) \cdot \tilde{y}(i,j-n) \end{aligned} \tag{4.2}$$

A disadvantage of filters separable along the horizontal and vertical dimension is that only orthogonal sampling pattern and not quincunx sampling pattern are supported.

The decimation process at the band splitting filters suppresses samples which have been determined by the expensive filter process. This is not very efficient. By implementing the filters in polyphase structures only partial results which are needed for the transmitted samples are determined. For polyphase filter structures the filter is split into several parallel filter parts. The impulse response of the filter parts are determined by a subsampled impulse response of the complete filter. If the number of phases corresponds to the decimation factor, each part of the filter processes only a certain phase component of the subsampled signal. By transition of the decimation process from filter output to input, the filter phases can operate with reduced clock frequency without affecting the function. This is depicted for a 2:1 decimation filter in Fig. 4.1. In this specific case the filter phases are given by filter parts with odd and even index respectively. The polyphase structure operates with the output sample rate which reduces the computational rate by a factor of two. Fig. 4.2 shows the filter structures for separable 2D filters with decimation of 2 in both dimensions. At the synthesis filter bank a similar polyphase structure reduces also the required computations. Instead of the insertion of zeros at the input a selection switch at the output of the polyphase filters is needed.

Filter banks of separable filters can be implemented by low pass and high pass filters. For QMFs the high pass and low pass filters have fixed relations on their coefficients. The transfer function H_1 of the high pass is a mirror image of the low pass filter H_0.

$$H_1(z) = H_0(-z) \tag{4.3}$$

With (4.3) follows for FIR filters

$$\begin{aligned} H_0(z) &= a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + \dots \\ H_1(z) &= a_0 - a_1 z^{-1} + a_2 z^{-2} - a_3 z^{-3} + \dots \end{aligned} \tag{4.4}$$

The filter characteristics G_0, G_1 for synthesis at the receiver can be directly derived from the filter H_0, H_1 at the transmitter.

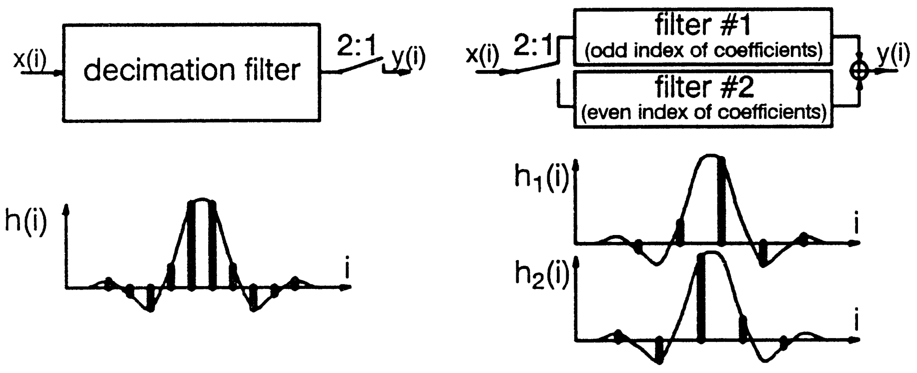


Fig. 4.1. Principle of polyphase filter structure for decimation

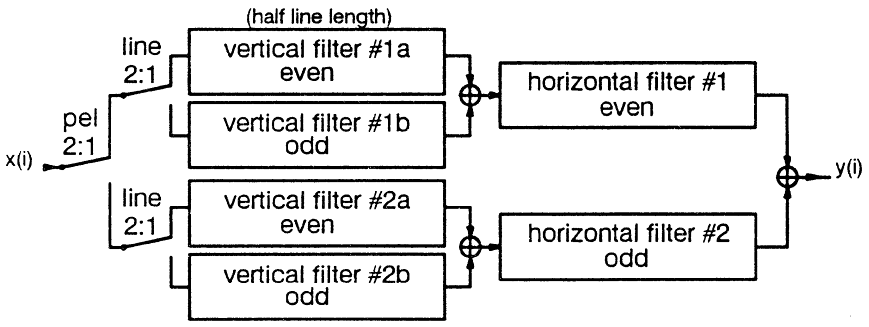


Fig. 4.2. Separable 2D decimation filter with combined polyphase subsampling

$$\begin{aligned} G_0(z) &= H_0(z) \\ G_1(z) &= -H_1(z) \end{aligned} \tag{4.5}$$

The filter coefficients for high pass and low pass filters of QMF filter banks differ only in the sign of every second coefficient. This can be utilized for implementation. The filter functions of the high pass and low pass filter can be split into two partitions. The regular change of the sign in (4.4) results in a structure (Fig. 4.3) similar to polyphase filters. According to the sign change, filter taps with even and odd indices are in separate partitions of the filter. The output values of the low pass filter are the sum of both filter partitions. Due to the QMF characteristics only one additional subtracter is necessary for realization of the high pass filter within a filter bank, since the difference of both partitions gives the output values for the corresponding signal. This reduces the hardware expense by nearly 50%. A similar structure exists for QMF synthesis filter banks (Fig. 4.4). For filter functions with an even number of taps the subfunctions of the filter parts # 1 and # 2 have the same coefficients but in reverse order. This results in a more regular design.

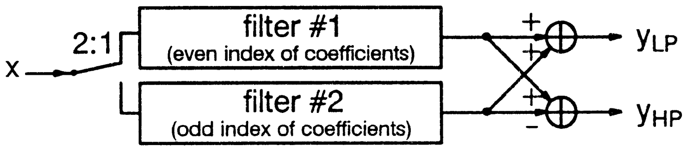


Fig. 4.3. Structure of a QMF decimation filter (analysis side)

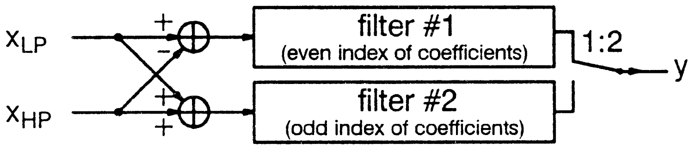


Fig. 4.4. Structure of a QMF interpolation filter (synthesis side)

The results received so far will be comprised for band splitting into 4 subbands by 2D filters with an impulse response $N \times M$. Taking 4 independent 2D filters for this filtering process the total computation rate becomes

$$R_c = 8 \cdot N \cdot M \cdot R_s \tag{4.6}$$

Considering separable filter characteristics and taking advantage of the special polyphase structure for QMF according to Fig. 4.3 the total computation rate will be reduced to

$$R_c = (N + M) \cdot R_s \tag{4.7}$$

For N and M being in the same order the reduction in computation rate is about $4N$. It has been proposed to use filters with 10 taps in vertical and 14 taps in horizontal direction [14]. For this particular example the computation rate of (4.7) is a factor of about 47 smaller than those of (4.6). Corresponding to this factor also the hardware expense of the computation part is smaller.

The main operations for filtering are multiplication and addition. The analysis above counts both just as one operation even if a multiplication has a much larger hardware expense than an addition. As a rough estimate one multiplier has the expense of n adders if n is the number of multiplicator bits. For video applications fixed point representation of the coefficients with 8 and 9 bit are required to fullfil the needed stopband attenuation. Considering the sign bit, 9 and 10 bit multipliers are needed. For a given filter characteristic the hardware expense for realization can be essentially reduced by an implementation with fixed coefficients instead of programmable filter coefficients. Fixed coefficients can be implemented by hardwired shifts and adders where the number of adders is specified by the number of nonzero binary digits. Figure 4.5 shows a typical impulse response of a FIR low pass filter. Just by considering a specific amplitude characteristic it is obvious that the coefficients apart from the center have a decay in magnitude which will result in several leading zeros in binary representation. The savings in number of adders by the decay of the coefficients magnitude is in the order of 50%. Integers can be represented by binary numbers with digits 0 and 1. As an extension signed digit representation is possible with digits -1, 0 and 1. Recoding of binary numbers to Canonical Signed Digits (CSD) provides a representation with at least 50% zeros [15]. From this follows that CSD coding of coefficients for typical low pass characteristics will result in at most 25% nonzero digits. The filter characteristics reported in [14] confirm this fact. The binary representation of these filters are listed in Tab. 4.1. For the specified coefficients approximately 20% nonzero digits are needed for representations. For implementation of the filter arithmetic additions, subtractions, and hardwired shifts of several samples are needed. Carry save adder trees are very appropriate for this. The sequence of additions can be either performed according to the sequence of coefficients or to the bit planes of all coefficients. By a bitplane grouping (least bitplane first) of the filter coefficients it is possible to have a nearly constant word length of all adders. Also early rounding and truncation of word length is supported.

TABLE 4.1
Coefficients and CSD representation of 14 and 10 tap filters [14], $\bar{1} = -1$

14 Tap Filter			10 Tap Filter		
Coefficients a_i	value	CSD	Coefficients a_i	value	CSD
a_6, a_7	247/512	$1\ 0000\ \bar{1}00\bar{1}$	a_4, a_5	248/512	$1\ 0000\ \bar{1}000$
a_5, a_8	48/512	$0\ 010\bar{1}\ 0000$	a_3, a_6	44/512	$0\ 010\bar{1}\ 0\bar{1}00$
a_4, a_9	-45/512	$0\ 0\bar{1}01\ 010\bar{1}$	a_2, a_7	-43/512	$0\ 0\bar{1}01\ 0101$
a_3, a_{10}	-3/512	$0\ 0000\ 0\bar{1}01$	a_1, a_8	-1/512	$0\ 0000\ 000\bar{1}$
a_2, a_{11}	12/512	$0\ 0001\ 0\bar{1}00$	a_0, a_9	8/512	$0\ 0000\ 1000$
a_1, a_{12}	-2/512	$0\ 0000\ 00\bar{1}0$			
a_0, a_{13}	-1/512	$0\ 0000\ 000\bar{1}$			

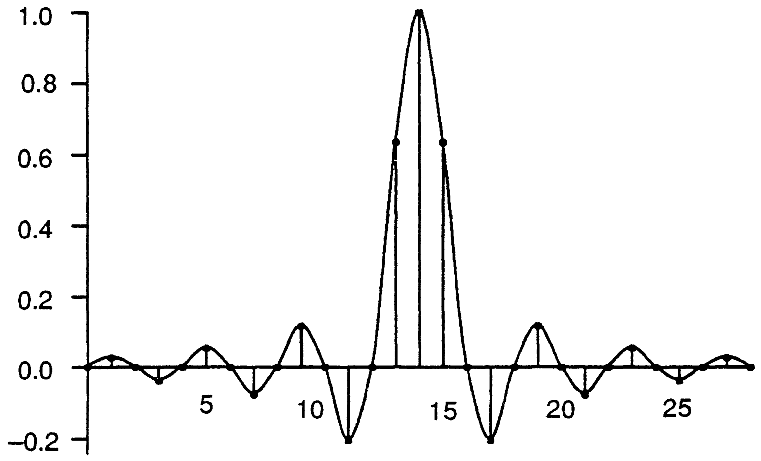


Fig. 4.5. Typical impulse response of an FIR low pass filter

4.1.2. I/O scheme and internal memories

The operational part of two-dimensional $N \times M$ filters have to be feeded in parallel with $N + M$ samples. In order to lower I/O bandwidth to a filter circuit, internal memories on chip are used. For filters along scanning direction (horizontal) a register chain according to the number of taps can provide all needed samples in parallel. For vertical filters a sequence of several line delays perform the same function. A disadvantage of this approach is the large amount of memory on chip. But the good news is that the I/O bandwidth is restricted to the video source rate. There are alternative architectures with an exchange between I/O bandwidth and internal memory. The amount of internal memory can be lowered for the expense of I/O bandwidth. The total delay needed is specified by the interface between video input and the filter operational part. A reduction of the internal memory will increase the capacity for external memory. Complete line delays on chip are frequently prefered because of simple control and direct feeding with the video sample rate.

The 2D filters are implemented as a sequence of two 1D filters. In order to avoid visible artifacts the results coming out of the first filter cannot be truncated to 8 bit. At least 12 bit representation for intermediate filter outputs is needed. Resulting from the large line length of 1920 active pels for HDTV signals, line delays form a major part of subband filter banks. For this reason the vertical filter with the line delays should be placed in front of the horizontal filter. In this case the data stored in the line delays are 8 bit words instead of 12 bit.

The line delays can be implemented by shift register arrays and RAM circuits.

The advantage of shift register arrays is the simple control but the disadvantage is the high power dissipation. By parallel implementation of shift registers and the use of a multiphase clocking scheme the power dissipation and silicon area can be considerable reduced [16]. The input multiplexers of the analysis filters (Fig. 4.2) result in a reduction of clock frequency in the filter sections. Horizontal down sampling by a factor of two requires the expansion of every second sample. In hardware realization this can be easily achieved by registers with a clock enable or by different clocking of registers. Vertical down sampling by a factor of two requires the expansion of the data of one line over the period of two lines. This kind of data formatting can be offered by FIFOs. The easiest implementation of FIFOs is possible with double sized memories and a read/write in a ping-pong mode. The memory capacity in the FIFOs can be lowered to the minimal amount by implementation as a rate conversion structure based on a combination of multiplexers and synchronous clocked shift registers of different length [16]. This kind of hardware structure for the input demultiplexer combined with the FIFO is depicted in Fig. 4.6.

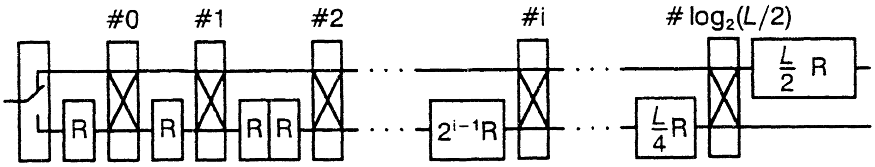


Fig. 4.6. Multistage synchronous polyphase FIFO unit

4.1.3. Prototype VLSI chip for subband filter banks

Prototype chips for a 10×14 subband filter bank have been designed for a 1.2μm two metal layer CMOS process. Because of the high transistor complexity of the filter bank an implementation with two identical chips has been considered. The application of the polyphase structure allows an easy partitioning of the filter bank into 2 identical chips. The HDTV signal is multiplexed twice at the input of the subband filter. This corresponds to the parallel polyphase structure of the architecture which is used to divide a filter bank into two parts.

Pel-by-pel multiplexed chrominance signals result in the same clocking rate and line length as for the luminance signal. Inserting in the horizontal filter part switchable delay parts one chip can perform operations for both luminance and chrominance. Two chips have been designed one as analysis filter the other as synthesis filter. Four identical chips form a complete filter bank for luminance and chrominance.

Fig. 4.7 shows the floor plan of both chip types. The line delays and FIFOs for vertical filtering and subsampling occupy the largest part of the chip area. Due to the dedicated filter arithmetic with multioperand adders, the arithmetic units are relatively small in terms of chip area. The block horizontal filter includes memory and arithmetic part. Both chip types have about 450 000 transistors on 90 mm^2. They have been manufactured and tested. Future CMOS technologies with smaller geometry will allow one chip realization of one filter bank.

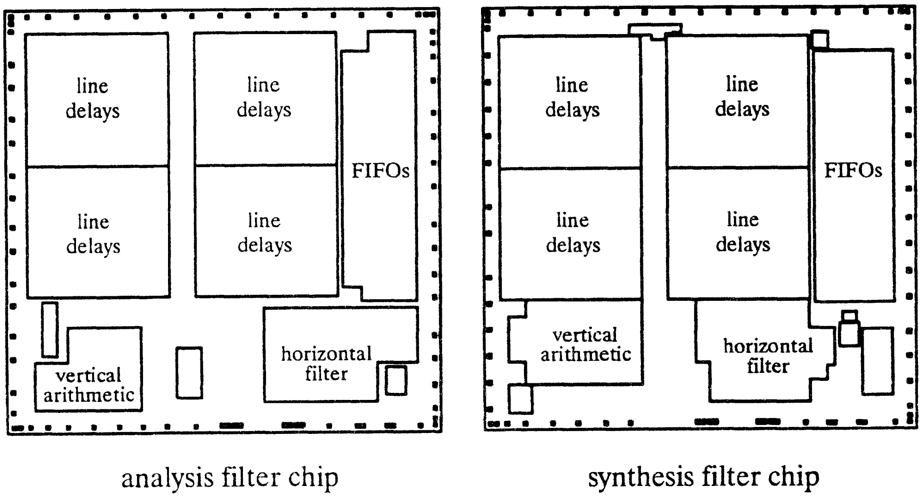


Fig. 4.7. Floorplans of analysis and synthesis filter bank ICs

4.2. Discrete Cosine Transform (DCT)

The purpose of transform coding is to convert a sequence of statistically dependent samples into an array of relatively independent and energy compacted coefficients. Because of high energy compaction, the DCT is the most frequently applied transform coding scheme. A DCT circuit has to perform continuously the transformation of blocks $\mathbf{X}$ of $N \times N$ image samples to $N \times N$ transform coefficients $y(u,v)$. In analogy to a Fourier series expansion of periodic functions the block $\mathbf{X}$ can be represented as a weighted sum of basis functions. In the Fourier series the basis functions are cos($\cdot$) and sin($\cdot$) functions and the weights are determined by scalar products of the original function with the basis functions. For 2D transformations the basis functions are described as basis images. Each block $\mathbf{X}$ can be represented as a linear combination of a set of basis images $\mathbf{\Phi}_{uv}$ weighted by the transform coefficients $y(u,v)$.

$$\mathbf{X} = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} y(u,v) \cdot \mathbf{\Phi}_{uv} \tag{4.8}$$

The transform coefficients $y(u,v)$ are determined by the dot product between $\mathbf{X}$ and $\mathbf{\Phi}_{uv}$. The dot product is a 2D scalar product.

$$\begin{aligned} y(u,v) &= \mathbf{X} \odot \mathbf{\Phi}_{uv} \\ &= \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x(i,j) \cdot \phi_{uv}(i,j) \qquad u,v = 0,1,\ldots,N-1 \end{aligned} \tag{4.9}$$

The DCT implementation according to (4.9) requires N^2 dot products between the block $\mathbf{X}$ and basis images. This counts to N^4 multiplications and additions. Also storage of N^2 basis images of size N^2 is needed. This shows that DCT implementation according to (4.9) is not very efficient considering that the basis images $\mathbf{\Phi}_{uv}$ can be determined as a product of two basis vectors.

$$\mathbf{\Phi}_{uv} = \mathbf{\Phi}_u \cdot \mathbf{\Phi}_v^T \tag{4.10}$$

N basis vectors $\mathbf{\Phi}_u$ form a $N \times N$ matrix $\mathbf{C}$. Therefore the DCT can be alternatively rewritten by the following matrix product.

$$\mathbf{Y} = \mathbf{C} \cdot \mathbf{X} \cdot \mathbf{C}^T \tag{4.11}$$

This can be seen as a sequence of 1D transform, transposition of N^2 intermediate results and a second 1D transform.

$$\mathbf{Y} = \mathbf{C} \cdot (\mathbf{C} \cdot \mathbf{X}^T)^T \tag{4.12}$$

The matrix multiplication according to (4.12) requires $2N^3$ multiplications and additions. Thus the computational effort of (4.12) is smaller by a factor $N/2$ when compared with (4.9). Also the storage requirements are reduced. A memory for N^2 elements of $\mathbf{C}$ and a transposition memory of size N^2 is needed.

A VLSI realization according to (4.12) has been proposed by Totzek et.al. [17]. A basic processing element (PE) in this structure has to contain a multiplier followed by an adder for accumulation. Employing the idea of a linear systolic array for implementation of the matrix-vector multiplication a high speed circuit with high regularity is possible. The multiplications and accumulations can be realized by carry-save adder trees. Intensive pipelining allows high clock rates. The devised chip mainly incorporates two linear arrays with 8 PEs each, a ROM for the coefficients of the matrix C and a RAM block for the transposition of an 8×8 block (Fig. 4.8). The computational part of this chip consists of 16 multipliers and accumulators. In a 1.5μm CMOS technology the IC requires a silicon area of about 92 mm^2 and contains 284 000 transistors. The achievable maximal sample rate is at least 45 MHz.

As an alternative to multiplier based realizations a distributed arithmetic can be used where partial precalculated results are stored in ROMs. Let the transformation of a vector $\mathbf{x}$ containing N values into a vector $\mathbf{y}$ given by

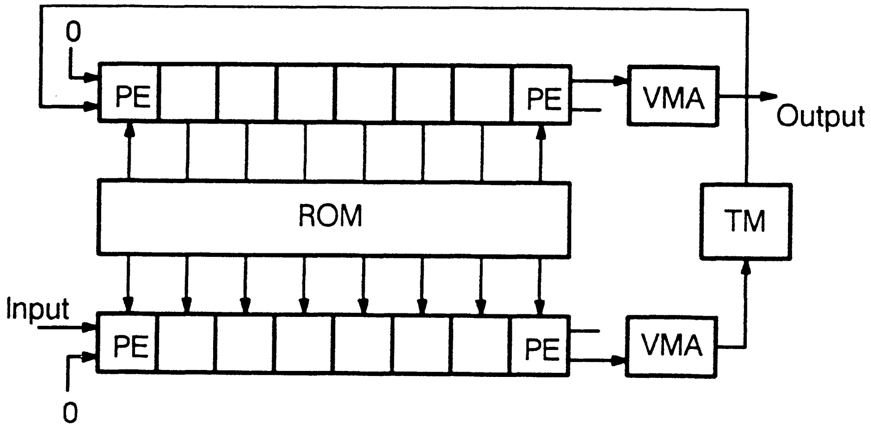


Fig. 4.8. DCT circuit based on matrix-vector multiplication.
VMA = vector merging adder, TM = transpositiiton memory

$$\mathbf{y} = \mathbf{C}\mathbf{x} \tag{4.13}$$

Considering a 2's complement code with m bits each vector **x** can be described by a sum of bit plane vectors $\mathbf{x}_r$.

$$\mathbf{x} = -\mathbf{x}_{m-1}2^{m-1} + \sum_{r=0}^{m-2} \mathbf{x}_r 2^r \tag{4.14}$$

Splitting the matrix **C** into N row vectors $\mathbf{c}(n)$ each value $y(n)$ has to be determined by

$$\begin{aligned} y(n) &= \mathbf{c}(n)\mathbf{x} \\ &= -\mathbf{c}(n)\mathbf{x}_{m-1}2^{m-1} + \sum_{r=0}^{m-2} \mathbf{c}(n)\mathbf{x}_r 2^r \end{aligned} \tag{4.15}$$

Let F_n be the scalar product between the row vector $\mathbf{c}(n)$ and a column vector $\mathbf{x}_r$. The precalculated results of F_n can be stored in a ROM which will be addressed by the bit patterns of bit plane vectors $\mathbf{x}_r$. This allows to design a PE as depicted in Fig. 4.9 which generates the values $y(n)$ based on table look-up by a ROM and bit serial processing. A 2D transform require $2N$ ROMs and a transposition RAM. For $N = 8$ the number of transistors for a ROM implementing the function F_n is in the order of that of a multiplier and accumulator. For this reason the chip size for $N = 8$ is for the distributed approach in the same order as for the matrix-vector implementation based on multipliers. The throughput rate for the distributed approach is smaller because the number of cycles depends on the word width m and not on N. The number of transistors for the ROM is growing exponentially by 2^N and not linearly with N. For this reason special additional measures to reduce the number of transistors for table look up are reported in [18]. The idea of the distributed arithmetic has been implemented in a chip performing 16×16 DCT for video signals with up to 15 MHz [18].

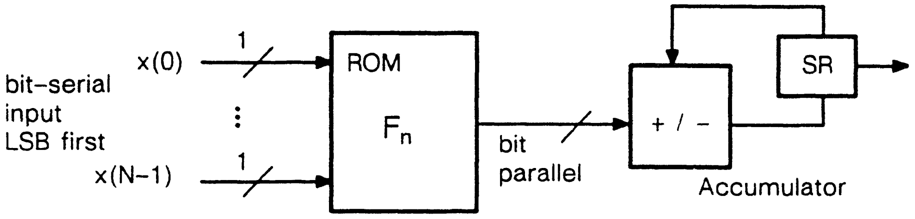


Fig. 4.9. PE for DCT based on distributed arithmetic

By consideration of the special characteristic of the coefficients of the matrix C architectural structures can be derived with reduced computational and storage requirements. The basis vectors of the 1D DCT are given by

$$\begin{aligned} \phi_u(i) &= b(i)\cos\left[(\pi/N)(i+1/2)u\right] \qquad & i = 0,1,\ldots,N-1 \\ & & u = 0,1,\ldots,N-1 \\ b(i) &= \begin{cases} \sqrt{1/N} & i = 0 \\ \sqrt{2/N} & i = 1,\ldots,N-1 \end{cases} \end{aligned} \tag{4.16}$$

By taking advantage of the periodicity in the basis vectors more efficient algorithms with smaller number of multiplications can be derived. These structures are refered to as Fast Cosine Transform (FCT). Lee [19] has proposed an algorithm that requires $N/2 \log N$ multiplications and $(3N/2)\log N - N + 1$ additions to perform a 1D DCT. A signal flow graph (SFG) of the FCT according to Lee is shown in Fig. 4.10. It contains an alternating sequence of data permutation stages and arithmetic stages. In an arithmetic stage the basic operation includes addition and subtraction followed by a multiplication.

A direct realization of the SFG would need an excessive amount of silicon area even if fabricated in an advanced technology. However, realization examples can be derived by using projections within the graph leading to sufficiently small implementations. Arterie et.al. [20] have presented a realization that uses a projection in the direction of wordlength to achieve a smaller silicon area. The realized chip mainly consists of a transposition memory, a parallel to serial and a serial to parallel converer, and an operative part. The operative part is a direct mapping of the SFG

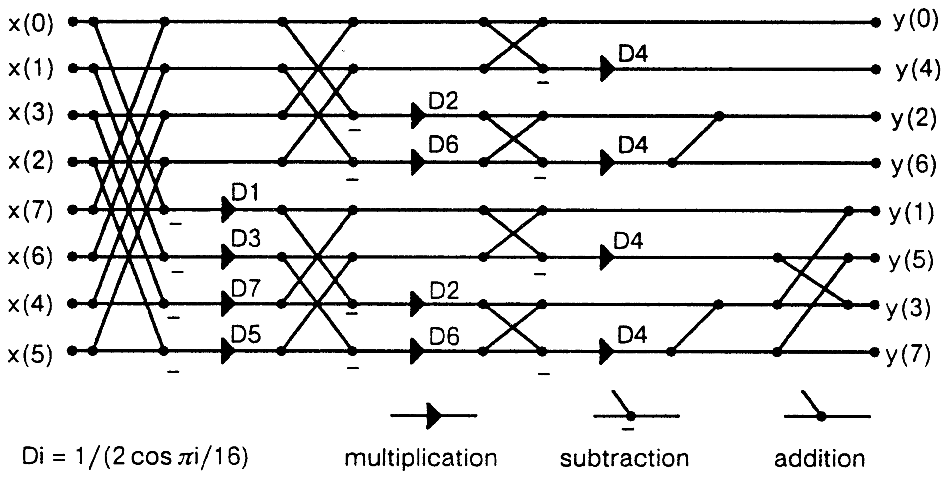


Fig. 4.10. SFG of the FCT based on Lee [19]

of the FCT into silicon and employs dibit serial techniques. Thus every addition, subtraction, and multiplication is assigned to a physical operator that receives its operands as a series of two-bit groups. A chip has been developed which supports 2D DCT and IDCT with $N = 4$, $N = 8$ and $N = 16$ for sample rates up to 13.5 MHz [20]. It has been fabricated in an 1.25μm CMOS technology and contains about 114 000 transistors in a die area of 40 mm^2. The advantage of a hardware structure according to the SFG of Fig. 4.10 is the reduced expense for the computational part and internal memory. A disadvantage is the complex data access which results in large amount of silicon area for interconnect.

Besides the reduction of the hardware expense by a transfer from bit parallel techniques in the direction of bit serial techniques also spatial projection techniques can be applied. A vertical projection in the SFG of Fig. 4.10 can also be employed. Through vertical projection, operations are mapped to PEs, called butterfly PEs, that generate two result values from two input values. One result is the sum of the two input values while the other result is their difference multiplied by a coefficient. Between the PEs a delay and commutator network is needed for the shuffling of data sequences. After the vertical projection a subsequent horizontal projection can be envisioned to generate a realization example with one PE (Fig. 4.11). A multiport RAM is needed for the data access and storage in the predefined sequence. Besides butterfly PEs also rotator PEs can be used as a basis element for DCT implementations. The rotator PE is more complex than a butterfly PE because four multipliers and two adders are needed. But the number of rotators is smaller. Ligtenberg and O'Neill [21] have reported on a DCT chip based on one rotator.

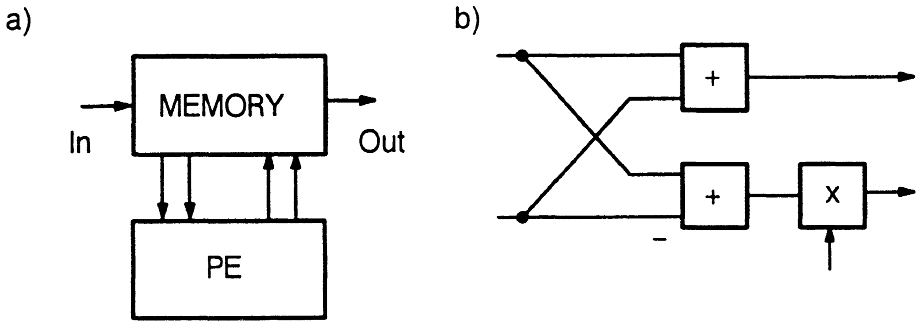


Fig. 4.11. a) DCT realization based on one PE b) butterfly PE

The projection techniques allow to adapt the number and type of PEs to the computational requirements by offering the right amount of concurrent processing. A DCT implementation with one PE is appropriate for low source rates as the CIF format. Broadcast video rates need at least 4 concurrent operating PEs. Combinations of the discussed schemes have been also implemented. The first stage of the fast algorithm (additions and subtractions) has been combined with proceeding parallel inner product stages of multipliers and accumulators [10]. Others combine the first stage of the fast algorithms with the distributed arithmetic [11, 18]. Combinations of the DCT with subtracters for prediction error determination and adders for sample reconstruction as needed in prediction loops have been also reported [10,11].

4.3. Block Matching Algorithm

Motion estimation is required to improve prediction of moving objects. Block matching is a simple scheme where motion is determined for small rectangular objects. Hereby the actual frame is divided uniformly into reference blocks with size $N \times N$ pels. Every reference block is compared with candidate blocks from a search area in the previous frame. The offset between the best matching candidate block and its reference block specifies the displacement vector $\mathrm{v} = (v_i, v_j)$. In general the mean of the absolute differences is used as a matching criterion. The search can be limited to a maximum displacement p in both directions if the maximum motion of objects is assumed to be limited (Fig. 4.12). The block matching algorithm is then given by

$$s(m,n) = \sum_{i=1}^{N}\sum_{j=1}^{N} |x(i,j) - y(i+m, j+n)| \qquad |m|, |n| \leq p \qquad (4.17)$$

$$u = \min_{m,n}\{s(m,n)\} \tag{4.18}$$

$$\mathbf{v} = (m,n)|_u \tag{4.19}$$

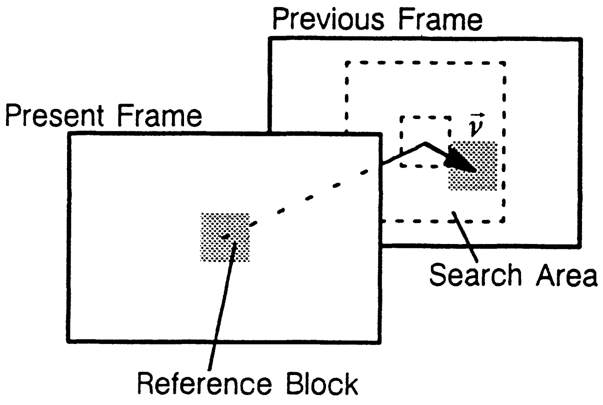


Fig. 4.12. Motion estimation based on block matching

The block matching algorithm requires the calculations of $(2p+1)^2$ sums according to (4.17) and a subsequent detection of the minimum sum to estimate the displacement vector from a full search of all candidate blocks. The computational rate is directly proportional to the number of candidate blocks. Full search considers $(2p+1)^2$ candidate blocks. Search strategies offers an essential reduction of the computational rate by reducing the number of investigated candidate blocks. The disadvantage of the search strategies is the enlarged control overhead and a much less regular data flow. For this reason realized block matching chips are based until now on full search techniques.

Realization of the block matching algorithm requires two types of basic PEs. Both are depicted in Fig. 4.13. One basic PE generates the absolute difference between two pels x and y from reference and candidate block, respectively, which is then accumulated for all N^2 pels within a block. Fig. 4.13a exemplifies an efficient implementation of the magnitude operation by using the MSB of the difference value to trigger bitwise inversion in the XOR operator as well as incrementation in the subsequent adder. In a second basic PE, the minimum sum is searched among the accumulated sums $s(m,n)$ and the corresponding displacement vector is detected. The first type of PE will be denoted as AD (absolute value of differences and accumulation) and the second as M (determination of minimum). The required processing power for real-time implementation can be achieved by a certain number of PEs of type AD and of type M. The numbers of PEs type M can in principle be smaller than the number of PEs type AD. Based on the computation rate the ratio

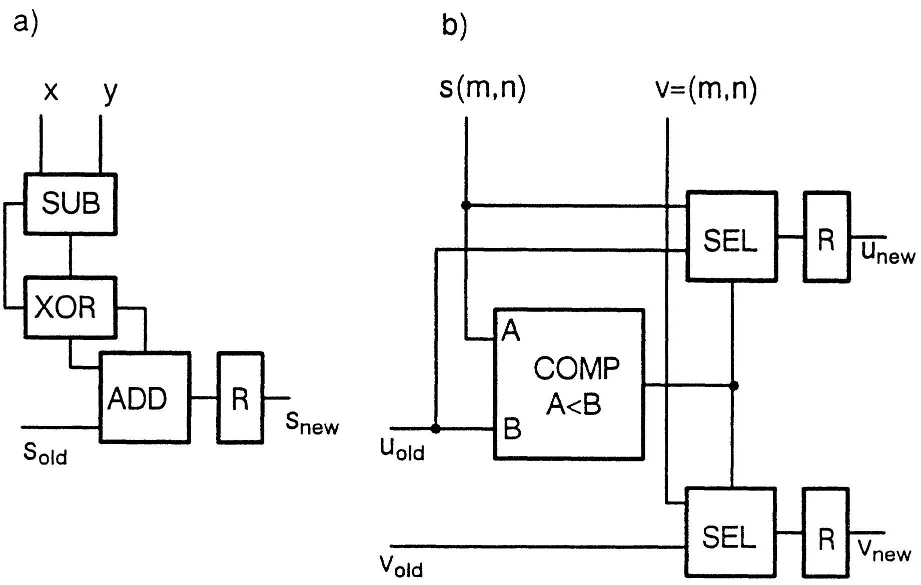


Fig. 4.13. Basic PEs for the block matching algorithma) PE type AD b) PE type M

could be $1 : N^2$. According to (3.1) the total numer of PEs should be in the order of

$$n_{PE} = (2p+1)^2 \cdot R_s \cdot T_{PE} \tag{4.20}$$

with T_{PE} as processing time of one PE. Because motion estimation is determined by the luminance component allone, the source rates given in Tab. 3.1 have to be reduced accordingly. The evaluation of (4.20) indicates that an extensive concurrency by pipelining and parallel processing is needed. Systolic architectures are very appropriate for this. Because of the associativity of additions and minimum search there are several alternative arrangements of the PEs to perform the calculations. A systematic design has to follow a methodology similar to those of Kung [22]. The DG of the algorithm has to be specified. By assignment of a schedule and multiple projections several alternative SFG can be derived [23]. Fig. 4.14 shows a 1D systolic array which offers the computational power for the CIF image format with reduced frame rate. The CCIR image format will require 2D arrays with N^2 or even $N \times (2p+1)$ PEs of type AD [23]. The systolic arrays require an adequate data transport to the boundaries of the array. A total of $2N^2 \times (2p+1)^2$ pels from reference block and search area have to be transported into the array during calculation of one displacement vector. The corresponding data rate cannot be transferred across the IC boundaries due to a limitation of pin count. In order to reduce the I/O rate at the IC boundaries, local memories have to be considered. Having stored the reference block and the search area in two local memories operating as double sized buffers in a ping-pong mode the I/O rate can be restricted to $3R_s$.

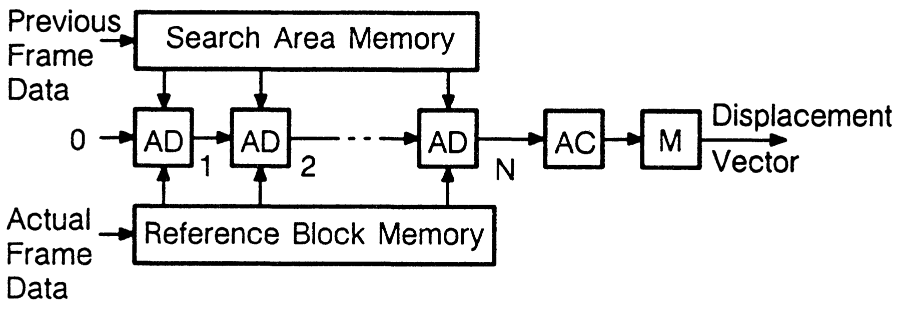


Fig. 4.14. 1D systolic array including local memory forrealization of block matching

In the literature several designs for block matching chips are reported [10, 11]. Frequently 1D processor arrangements with local memories based on register arrays according to the proposal of Yang et.al. [24] have been applied.

5. Programmable Multiprocessor System

A software oriented implementation is very attractive because it incorporates flexibility to accommodate a wide variety of application schemes and it allows modifications of algorithms by software changes. In order to improve the overall silicon efficiency special multiprocessor architectures have been developed which incorporate data path and data access adapted to the required class of algorithms. With respect to data and control flow, architectures of multiprocessors are generally classified as single instruction, multiple data stream (SIMD) or multiple instruction, multiple data stream (MIMD). Block diagrams of generic SIMD and MIMD architectures are shown in Fig. 5.1. An interconnection network is provided for communication between the processing units. In order to restrict the I/O bandwidth for operand transport, memories local to the processing units aree needed for video applications. The appropriate size of the local memories depends on the kind of algorithms and is driven by multiple access to image data of local image segments. Because most of the image processing algorithms allow independent processing of image segments the PEs will be distributed over the image space (see section 3). In this case image data have to be distributed to the PEs according to a specification of image segments. The need for communications between the PEs can be avoided by matching the local memory size to the maximum requirements of the set of algorithms to be considered.

SIMD multiprocessors are very efficient for applications where the employed algorithms allow identical operations for many parallel data streams. Most low level algorithms with fixed time dependent processing sequence are of this kind.

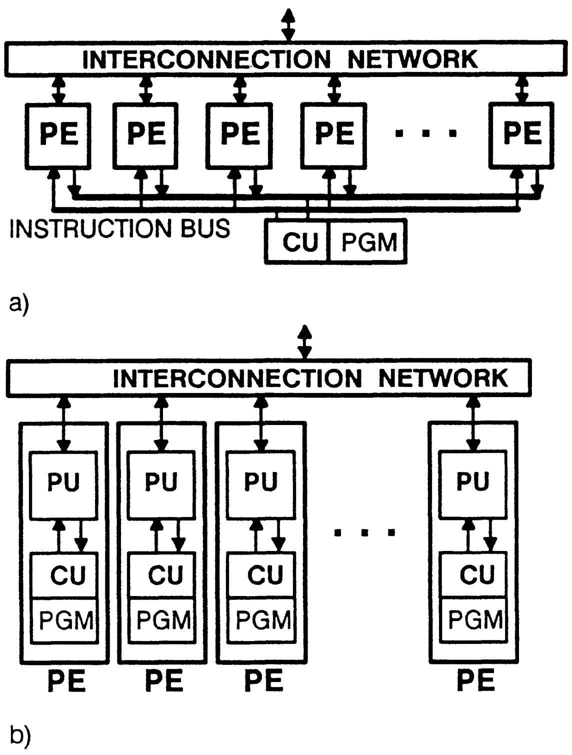


Fig. 5.1. Architectures of SIMD (a) and MIMD (b) multiprocessors
PU = processing unit, CU = control unit, PGM = program memory

The efficiency of SIMD decreases for algorithms with result dependent alternative sections. During operations of alternatives, parts of the PEs have to be disabled and idle, while the remaining PEs perform the operations. This loss of efficiency for data dependent processing can be reduced for MIMD architectures because independent processing of all PEs is possible.

Multiprocessor systems can be built of cascaded processor clusters where each cluster is matched to subsets of the algorithms. A frequent proposal is to use processor clusters assigned to low level, medium level, and high level algorithms. The number and type of processors considers the particular requirements of each level. Instead of the sophisticated systems based on processor clusters in the following linear arrangements of bus connected processors will be discussed because of simplicity.

5.1. Improvement of multiprocessor performance

The design goal is to achieve system implementation with smallest hardware complexity. Besides scalibility and regularity of systems, the overall silicon area can be taken as a design criterion. The silicon area of the total multiprocessor system is influenced by the performance efficiency which is the quotient of achievable performance and peak performance. For performance efficiency the efficiency in terms of computation rate and access rate is of particular interest.

The efficiency of the access rate and the size of local memories depends on the data transfer concept. Two loading schemes for bus connected multiprocessor systems are depicted in Fig. 5.2. The first considers dual-ported local memories. During parallel processing of all PEs the data of the following image segments are loaded. Double sized local memories with independent read/write access are needed. The second scheme requires only single-ported local memories. Immediately after loading of one local memory the assigned PE starts processing. This kind of loading scheme is only possible for MIMD controlled processors. Non-overlapped loading and computation is not very efficient for SIMD controlled processors. During loading of one local memory all other PEs idle. Processing can start after having loaded all local memories. The discussion above shows that the MIMD controlled PEs have the advantage of half size local memories in case of non-overlapped loading. But for the silicon efficiency it has to be considered that MIMD is more complex because of its own controller and program memory.

The computation rate of the PEs can be increased by considering the type of operations and the inherent parallelism of the algorithms. The number of concurrent operations within a PE can be increased by pipelining and parallel data paths. A PE with increased parallelism in the data path is displayed in Fig. 5.3. This kind of PE is very appropriate for low level algorithms as filtering, transformation, and block matching. The offered concurrency cannot be utilized for medium and high level algorithms. For this reason separate additional processing units are proposed. As a result the effective computation rate is algorithm dependent. The discussed sophisticated PE structures are of advantage as long as the increase of effective computation rate is larger than the increase of silicon area.

The literature reported about several multiprocessor systems for video signal processing [25, 26, 27, 28, 29]. The structures, basic elements, and technologies are different. For this reason the overall performance and the needed silicon area using advanced technologies are very difficult to compare. A first attempt to achieve a unified performance measure for multiprocessor architectures independent of technology constraints have been investigated in [30].

5.2. Design Examples

Two design examples of multiprocessor systems will be discussed here. One based on a SIMD architecture, the other is a sophisticated MIMD architecture.

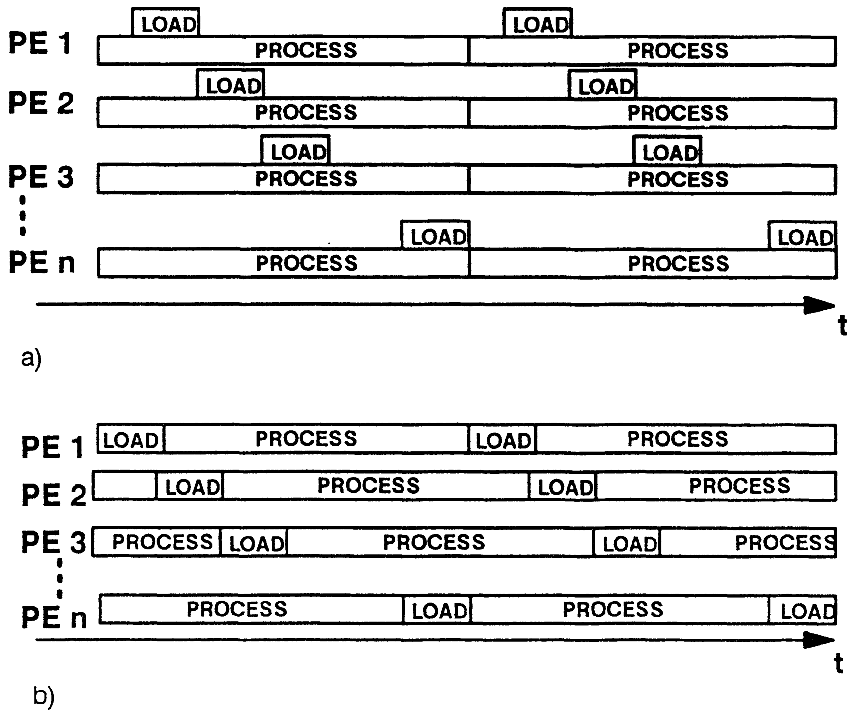


Fig. 5.2. Loading schemes for multiprocessor systems
a) Overlapped loading and computation with dual-ported local memories
b) Non-overlapped loading with single-ported memories

A multiprocessor IC with SIMD architecture has been proposed by Micke et al. [25]. One multiprocessor IC contains an I/O unit for video data, a sequencer with instruction decoder or access to instructions in an external program memory, and $n = 6$ identical PEs. A block diagram of the PE is shown in Fig. 5.4. A PE consists of local memory with 512×16 bit RAM and a 16×16 bit register file, and a processing unit for operations on 16 bit operands. Special pipeline implementation with serially connected ALU, multiplier, accumulator and shifter supports arithmetic operations as

$$\sum |a - b|, \quad \sum (a - b)^2 \quad \text{or} \quad \sum (a - b) \cdot c. \tag{5.1}$$

Pipelining within the arithmetic unit offers up to four operations in one cycle. The PEs are connected via an interprocessor communication link that allows left or right shift of data along the PE string. Input and output of data are performed over a data bus common to all PEs within the chip.

The described multiprocessor IC has been realized in a 0.7μm CMOS technology. A clock frequency of 25 MHz is achieved and this supports a peak processing

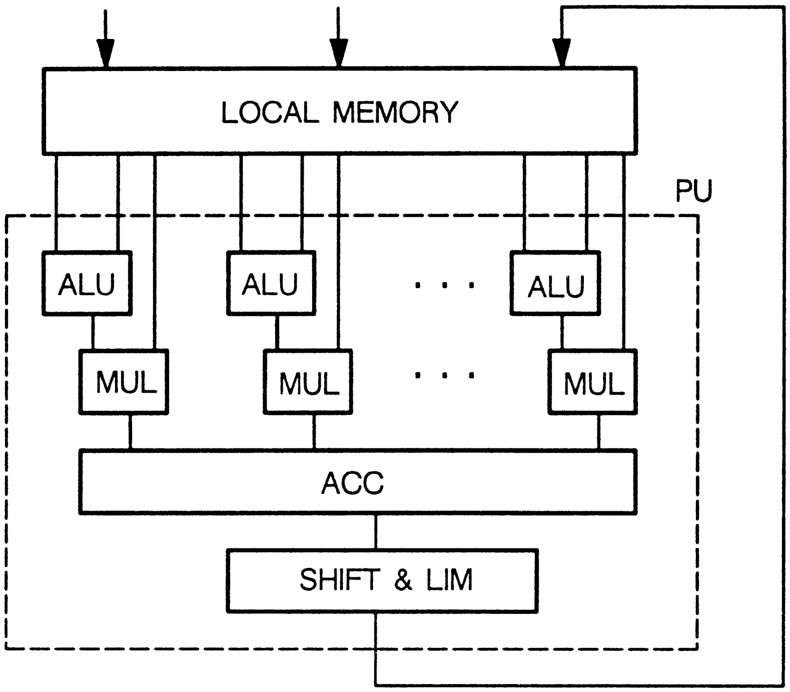


Fig. 5.3. Processing element with increased parallelism in the data path

rate of 150 M accumulations per second. In terms of arithmetic operations this amounts to a peak rate of 600 MOPS per chip. For applications requiring higher processing power a parallelization of several multiprocessor chips is possible. A system containing several multiprocessor chips on a printed board has been proposed [25] for implementation of the hybrid coding scheme for video phone applications. In this system each multiprocessor chip has its private external buffer memory. One of the parallel operating multiprocessor chips performs the overall system control providing its sequencer for addressing of instructions in a single program memory.

The complete system consisting of 48 PEs supplies a peak processing power of 4800 MOPS. The reported implementation offers hybrid coding of CIF video signals with 12.5 Hz frame rate. The hybrid coder and decoder functions require a total of about 600 MOPS in case of 12.5 Hz frame rate. Thus the system operates with an efficiency in the range of 10 to 15 % for this application.

A higher efficiency can be achieved by MIMD architectures which allow independent processing of the individual PEs. A MIMD multiprocessor architecture adapted to the requirements of video coding is reported in [27]. The PE architecture

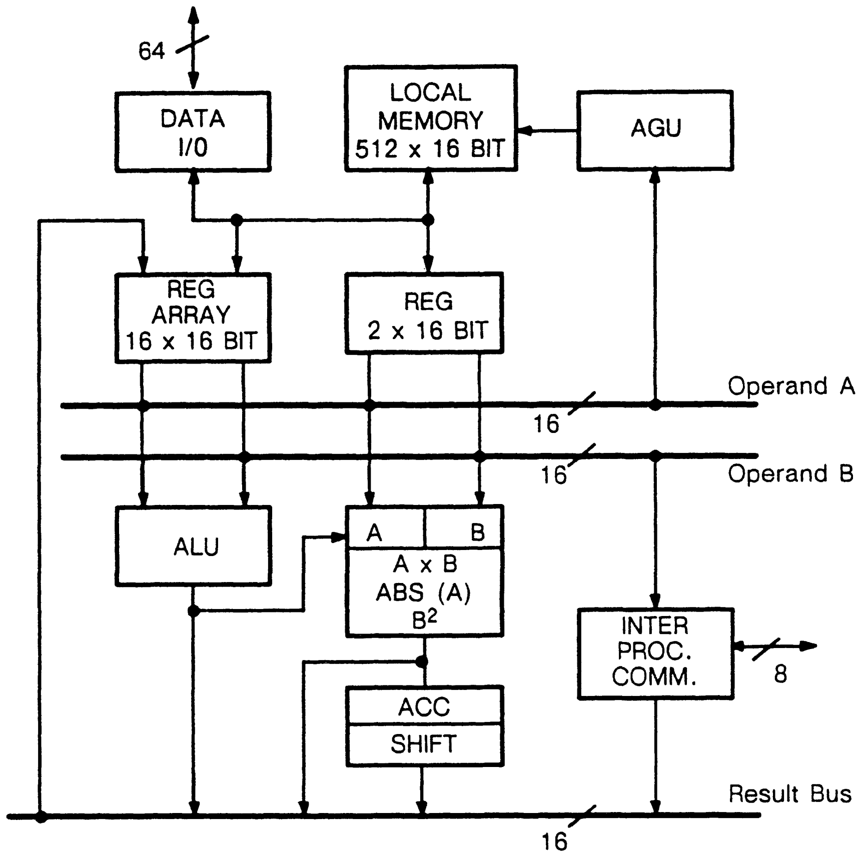


Fig. 5.4. PE of a SIMD based multiprocessor chip [25]

is depicted in Fig. 5.5.

A PE contains local memory (LM) for storage of source data and intermediate results, an address generation unit (AGU), an arithmetic processing unit (APU), a medium level processor (MLP), and a programm memory (PGM). The proposed size of the local memory is 512 $\times$ 32 bit. In order to have high flexibility a micro controlled AGU is considered. The AGU is generating address sequences of the LM. Furthermore, controlling of the APU is provided by the AGU. It is microprogrammable and supports the processing of a sequence of tasks, e.g. motion estimation followed by FIR filtering and 2D DCT. The proposed APU is adapted to window based low level algorithms and can operate in parallel 4 data streams. It performs in pipelining a sequence of ALU-operations, multiplication, accumulation over a window, and shift and limit for normalization of sums. For

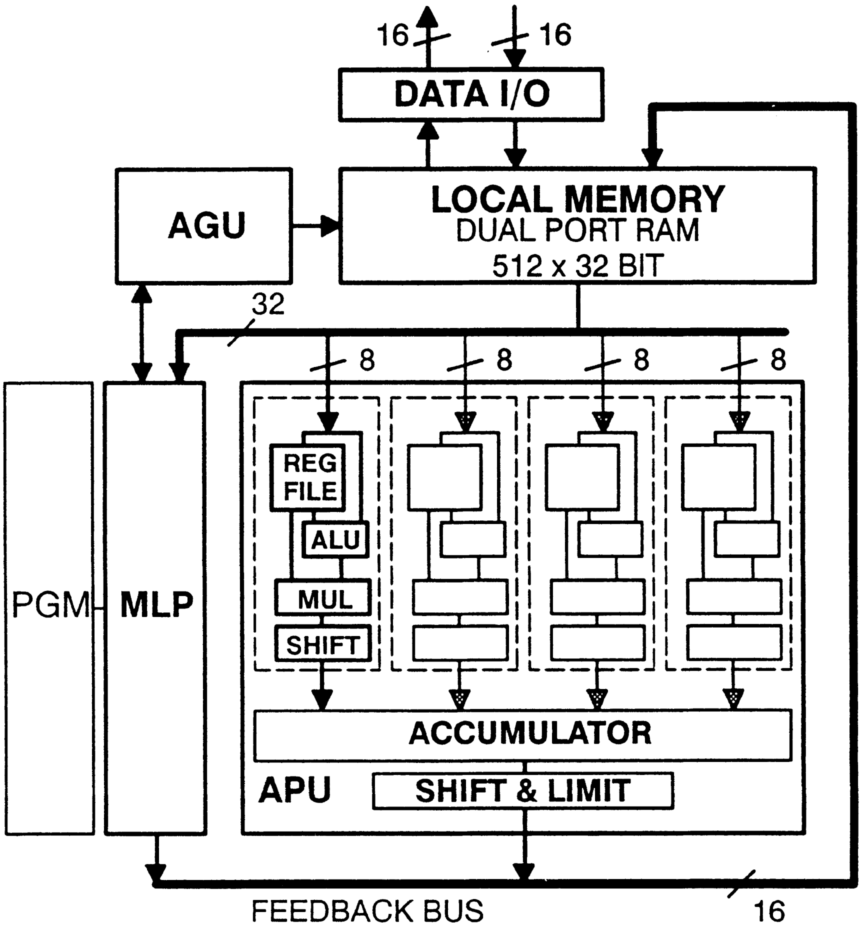


Fig. 5.5. PE of a MIMD based multiprocessor architecture [27]

the access to two operands a register file of 128 × 8 bit is included. The APU is based on data paths with 8 bit representation. Nevertheless, the mode shifters between multiplier and accumulator enable the APU to perform 16 bit operation in two consecutive clock cycles. During the first clock cycle the lower bytes of all operands are processed. Processing of the upper bytes of the operands is performed during the following clock cycle. High level operations, medium level operations, and down load of the microprogram for the AGU are processed by the medium level processor (MLP).

The MLP has memory mapped bidirectional access to the local memory and write access to the microprogram RAM of the AGU. A powerful RISC core can be employed as MLP. In case of 40 MHz clock rate a peak performance of 640 MOPS is achieved in the 8 bit mode. The computation rate is half for the 16 bit mode. Simulations of the architecture have shown that for video coding schemes a performance of 20 to 30 % of the peak computational rate should be achievable.

Comparison of the two presented multiprocessor architectures indicate that the first SIMD architecture based on simpler PEs offers to integrate several PEs on one chip whereas the more sophisticated MIMD architectures results in 1 PE per chip. But the overall performance for the second solution is higher because of the larger inherent parallelism and higher flexibility for processing. It has been estimated that for a hybrid video coding scheme 8 chips of the SIMD architecture have the same performance as 6 chips of the MIMD architecture.

5.3. Conclusion

Compact and cost effective realization of real-time video signal processing systems call for VLSI implementation. In order to cope with the high performance requirements architectural structures with extensive pipelining and parallel processing are requested. The VLSI implementation has to consider the trade-off between computation part, memory and I/O bandwidth. The most compact implementation can be achieved by VLSI circuits dedicated to special processing tasks. Hereby all a priori known features have to be incorporated into the architecture. For implementations with high flexibility for modifications, programmable systems are needed. The number of processing elements of SIMD and MIMD multiprocessor systems can be kept small by matching the architectural parameters to the requirements of considered algorithm classes.

References

[1] CCITT Study Group XV: Recommendation H 261, Video Codec for Audiovisual Services at px64 kbit/s, Report R 37, Geneva, July 1990
[2] CCIR Recommendation 601: Encoding parameters of digital television for studios; in Recommendations and Reports of the CCIR, vol. XI, pt.1 ITU 1982, Geneva, Switzerland
[3] CCIR: Draft new report AD/CMTT on the digital transmission of component coded television signals at 30-34 Mbit/s and 45 Mbit/s, CCIR-Document (1986-1990) CMTT/116
[4] H. Yasuda: Standardization activities on multimedia coding in ISO, Signal Processing Image Communications 1 (1989) pp. 3-16
[5] ISO-IEC JTC1/SC2/WG11 MPEG-90/176 Rev. 2, (1990)
[6] Draft revision of Recommendation H. 261: Video codec for audiovisual services at px64 kbit/s, Image Communication, vol. 2, no. 2, pp. 221-239, August 1990
[7] H.G. Musmann, P. Pirsch, H.J. Grallert: Advances in Picture Coding, Proc. IEEE, vol. 73, no. 4, pp. 523-548, April 1985
[8] A. Nagata, I. Inoue, A Tanaka, N. Takeguchi: Moving picture Coding system for digital storage media using hybrid coding, Image Communication, vol. 2, no. 2, pp. 109-116, August 1990
[9] H. Gharavi, A. Tabatabai: Sub-band coding of monochrome and color images, IEEE Trans. Circuits and Systems, vol. CAS-35, pp. 207-214, February 1988
[10] P.A. Ruetz, P. Tong, D. Bailey, D. Luthi, P. Ang: A high-performance full-motion video compression chip set, to be published

[11] H. Fujiwara, M.L. Liou, M.T. Sun, K.M. Yang, M. Maruyama, K. Shomura, K. Oyama: An all-ASIC implementation of a low bit-rate video codec, to be published

[12] R.E. Crochiere, L.R. Rabiner: Multirate digital signal processing, Prentice Hall (1983)

[13] J.J.T. Smith, T.P. Barnwell: Exact reconstruction techniques for tree-structured subband coders, IEEE Trans. on ASSP (June 1986), 34, No. 3, pp. 434-441

[14] U. Pestel, K. Grüger: Design of HDTV subband filterbanks considering VLSI implementation aspects, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 1, No. 1, pp. 14-21, March 1991

[15] K. Hwang: Computer Arithmetic, Principles, Architectures and Design, John Wiley & Sons, 1979

[16] P. Pirsch, K. Grüger, M. Winzker: VLSI architectures of two-dimensional filters for HDTV coding, to be published in conferencen records ISCAS'92, San Diego

[17] U. Totzek, F. Matthiesen, T. Noll: DCT-Bausteine für die Codierung von HDTV-Signalen, mikroelektronik, vol. 5, no. 3, pp. 124-127, 1991

[18] M.-T. Sun, T.-C. Chen, A.M. Gottlieb: VLSI Implementation of a 16x16 Discrete Cosine Transform, IEEE Trans. Circuits and Systems, vol. 36 (1989), no. 4, 610-617

[19] B.L. Lee: A New Algorithm to Compute the Discrete Cosine Transform, IEEE Trans. Acoustics, Speech, and Sig. Proc., vol. ASSP-32 (1984), no. 6, 1243-1245

[20] A. Artieri et al.: A One Chip VLSI for Real Time Two-Dimensional Discrete Cosine Transform, Proc. IEEE Int. Symp. Circuits and Systems, Helsinki (1988), 701-704

[21] A. Ligtenberg, J.H. O'Neill: A Single Chip Solution for an 8 by 8 Two Dimensional DCT, Proc. IEEE Int. Symp. Circuits and Systems, Philadelphia (1987), 1128-1131

[22] S.Y. Kung: VLSI Array Processors, Englewood Cliffs, NJ, Prentice Hall, 1988

[23] T. Komarek, P. Pirsch: Array architectures for block matching algorithms, IEEE Trans. on CAS, vol. 36, no. 10, pp. 1301-1308, October 1989

[24] K.-M. Yang, M.T. Sun, L. Wu: A family of VLSI designs for the motion compensation block matching algorithm, IEEE Trans. on CAS, vol 36, no. 10, pp. 1317-1325, October 1989

[25] Th. Micke, D. Müller, R. Heiß, ISON-Bildtelefon-Codec auf der Grundlage eines Array-Prozessor-IC, mikroelektronik, vol. 5, no. 3, pp. 116-119, 1991

[26] P. Weis, Video-Parallelprozessor zur Bild-codierung und -verarbeitung auf einem Chip, mikroelektronik, vol. 5, no. 3, pp. 112-115, 1991

[27] K. Gaedke, H. Jeschke, P. Pirsch, A VLSI based MIMD architecture of a multiprocessor system for real-time video processing applications, submitted to the Journal of VLSI Signal Processing

[28] Y. Suzuki et. al., Single board video codec using VLSIs for 64/128 kbit/s CIF video, Proc. of Int. Picture Coding Symposium, pp. 3-1, 1990

[29] T. Nishitani, Parallel video signal processor configuration based on overlap-save technique and its LSI processor element: VISP, Journal of VLSI signal processing, vol. 1, no. 1, pp. 25-34, 1989

[30] H. Jeschke, K. Gaedke, P. Pirsch, Multiprocessor performance for real-time processing of video coding applications, submitted to IEEE Trans. on CSVT

Compiler Techniques for Massive Parallel Architectures

LOTHAR THIELE
Institute of Microelectronics,
Universität des Saarlandes, Im Stadtwald, D-6600 Saarbrücken, Germany

Abstract. The paper is concerned with the design of massive parallel architectures. It contains an overview of existing compilation techniques. In particular, tools for a mechanical and provably correct design trajectory are described. New results concerning a hierarchical design methodology based on the class of piecewise linear programs are presented.

1. Introduction

The paper is concerned with parallel architecture design for massive real-time computations.

There is a strong relation between the *application domain* (e.g. image sequence processing, computational algebra, combinatorial optimization, real-time signal processing, computer graphics), *algorithm domain* (e.g. sequential vs. parallel computations, imperative vs. functional program, regular vs. complex data flow, static vs. dynamic parallelism, determinism vs. non-determinism) and the *architectural domain* (e.g. general purpose vs. application specific processor, synchronous vs. asynchronous communication, massive vs. large grain parallelism). For example, in a high-throughput application domain a general purpose sequential processor is not capable to satisfy the imposed real-time constraints. Even a parallel general purpose architecture (e.g. MIMD or SIMD) may not be appropriate because of the control overhead, the high area and power consumption and the restricted I/O bandwidth between processors, to input/output ports and to memory banks. On the other hand, if a regular array architecture (e.g. systolic array, wavefront array) is chosen, the algorithm to be implemented should match the regular communication structure and parallelism. This impact parallel algorithms and architectures have on each other can be efficiently used to design algorithms and architectures simultaneously. As a result, the architecture may reflect properties of the implemented algorithm and vica versa.

In the design of (parallel) computer architectures *software and formal languages* play an eminent role in many different aspects.

- It is generally accepted that CAD is necessary to master the complexity of designing architectures for high-throughput algorithms. Reasons for this assessment are the required fast turn around time, the iterative design strategy which necessitates a fast prototyping and the certifiable correctness of the design.
- Within the whole design trajectory there may be at some place a compiler involved whose primary inputs are a specification of the algorithm to be implemented and a partial specification of the target architecture. As a result,

a specification of software and hardware is obtained which specifies the hardware/software implementation of the given algorithm and which fulfills the input specifications. The class of algorithms and architectures the design system is intended for influences the form of input specifications, e.g. language vs. graphical/interactive, imperative vs. functional language.

- A formal design language is often used in order to guarantee that the stages of a design are consistent and formal verification methods can be used to proof correctness, i.e. to proof that at any stage of the design the given specifications are satisfied. A formal design language is also necessary if provably correct design methods are implemented in the design system.
- The path from input specifications to an architecture involves the solution of problems whose complexity depends on the chosen classes of algorithms and architectures. For example, it is well known that the use of general purpose parallel architectures suffers from the complexity of the corresponding compilation problems. If we restrict to the class of regular array processors, the mapping problems are comparatively goodnatured. Relations between architectural style and compilation problems can be found on all levels of an architecture, e.g. a specialized, non-orthogonal and restricted instruction set of a processor may lead to a simpler architecture but necessitates complex compilation strategies.
- Yet another trade-off is the partitioning of the architecture into programmable and specialized components. Consequently, software for programming general purpose subsystems may be part of the final result of a design process.

Finally, a remark similar to that at the end of the preceding paragraph is appropriate: the greater flexibility in designing software and hardware components simultaneously may lead to higher performance in comparison to restricting the design space.

It is useful to distinguish two aspects of design systems for signal/image processing architectures: *environment and tools.* The tools perform the transition from the input specification to the specification of the implementation, including the specification of hardware and software subsystems and the assignment of functions to processing elements. Consequently, the tools reflect the methodology based on consistent mathematical models and methods. The environment is responsible for the interfacing between the designer and the available methods and tools, for the data management and documentation of the design. Corresponding to this classification, the paper is organized as follows: Section 3 is devoted to different aspects of the environment, including requirements and a survey of existing implementations. In Section 4, a set of requirements for CAD tools is described. The basic methodology underlying the synthesis of piecewise regular architectures is presented in Section 5. The described methodology only covers the architectural design problem.

2. Targets

Referring to the above mentioned complexity of the architecture design problem and the necessity to exploit special properties of application domains and algorithm classes we are lead to the conclusion that there will be no unique design approach suited for all target classes of applications, architectures and algorithms. Therefore, at first the main scope of the paper should be defined.

2.1. Target Applications

The area of applications we are going to consider can be characterized by the need of massive real-time computations. Examples of potential application domains are *image (sequence) processing, computer graphics, computational linear algebra, real-time signal processing and seminumeric algorithms.* In the case of image processing, there is the necessity for high-throughput architectures if local operations on the given image sequence must be performed (e.g. 2 and 3 dimensional filtering, source coding, image restoration algorithms). As the corresponding algorithms operate at the pixel level, many identical operations on neighboring data are required. Consequently, the data flow exhibits modularity, regularity, locality and massive parallelism. Similar observations hold in the cases of matrix computations in algebra, computer graphics and some combinatorial or seminumeric algorithms.

2.2. Target Architectures

The paper is concerned with a target architecture that can be classified referring to a taxonomy used in [1] as follows:

Regular Array As the target application leads to massive real-time computations, the following constraints for the target architecture can be formulated: massive parallel computation, intensive multilevel pipelining, local interconnection scheme, distributed memory and computing power. Special classes suited for VLSI implementation are known as regular processor arrays, e.g. systolic or wavefront arrays [2, 3]. These arrays consist of a regularly connected mesh of identical processing elements that perform a time invariant processor function. It runs out, however, that a generalization of this model is necessary for a variety of reasons: (1) the algorithms to be implemented very often lead to a more complex interconnection structure which may be called piecewise regular [4, 5, 6], (2) even if regular algorithms are implemented, the consideration of finite resources and the consideration of input/output data leads to non-regular processor arrays and (3) the concept of piecewise regular processor arrays directly fits a hierarchical design methodology, see [7]. In particular, architectural design for the class of piecewise regular arrays is considered in this paper. The whole system can be partitioned into regular subsystems. Therefore, the target architecture is composed of a set of communicating regular arrays which consist of locally interconnected processing elements equipped with local computing power and local memory. Because of the piecewise regularity, there will be the possibility to also include the specification of external memory, of I/O processors and of external control processors in the design process.

Limited programmability The target architectures belong to the class of application specific processors. In order to reduce the control, area and power overhead the programmability of the individual processing elements exactly fits the given class of algorithms and applications. On the other hand, the proposed tools and methodologies can be also used to solve compilation problems for general purpose parallel processing systems.

Control Flow As well the restriction on an interconnection structure which is local in time and space as the piecewise regular architecture directly leads to a local control flow concept. Control data locally propagate through the system and determine the interconnection structure and individual functions of the processing elements.

Multilevel Pipelining and Parallelism The hierarchical nature of available design methodologies enables the consideration of internally pipelined processing elements. These elements may internally contain arithmetic pipelines or parallel arithmetic units. Consequently, this property can be used to introduce bit level pipelining, efficient bit-parallel or bit-serial processing elements.

In addition, the special importance of this class of target architectures is based on the fact that they can be designed automatically by provably correct and optimal synthesis methods, see e.g. [8, 9, 3].

2.3. Target Specifications

The design of architectures starts with a behavioral specification of the algorithm to be implemented. This formal specification of the required input/output map of the architecture is usually not directly related to the structure of the architecture. This information is added in the course of the design processes. For the description of the required system behavior either imperative or functional description styles are possible. One of the main characteristics of algorithms that can be mapped efficiently on the class of target architectures described above is their *static parallelism*. As parallelism is infused before run-time an efficient mapping of the algorithm on a target architecture can be determined at compile-time. Consequently, one does not have to trade-off the time spent to determine the parallelism at run-time and the expected savings by exploiting it. The close relation between algorithms and architectures necessitates a *regular data flow* imposed by the specification. In case of an imperative program, e.g. a *concatenation of nested loop programs* enclosing a body with local conditionals and affine dependences between the variables satisfies this requirement. In a functional style, algorithms with a (piecewise) regular data flow can be represented by a set of *quantified equations* involving linearly indexed variables. As a language describing different stages of a

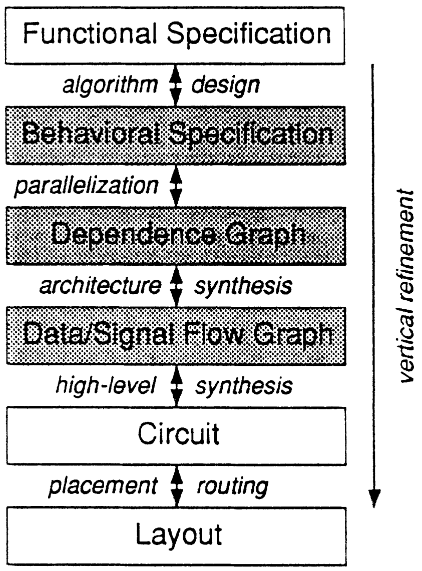


Fig. 1. A simplified design flow between levels of abstraction and vertical hierarchy. The marked boxes are the subject of the tools described in Section 5

design process, a functional specification is preferable, e.g. because of the inherent single assignment property, the type checking possibilities, the support given to guarantee a provably correct design, a direct relation between functions defined in the language and subsystems of a corresponding system, and the close syntactical and semantic relationship to usual mathematical expressions.

3. Environment

As the main topic of this paper are the tools necessary for designing architectures for high-throughput applications, we will look at the necessary design environment only briefly.

3.1. Design Trajectory

There are different views of the design trajectory for the above defined target architecture. In particular, we will distinguish between the design flow between different levels of abstraction *(vertical hierarchy)*, the design flow within a certain level *(horizontal hierarchy)* and a global view of an *interactive design trajectory*. In chapter 4, we will present a more detailed view of the design flow corresponding to the tools for architecture synthesis.

Fig.1 represents a simplified design flow between different levels of abstraction. The *functional specification* formally states the input/output map of the desired

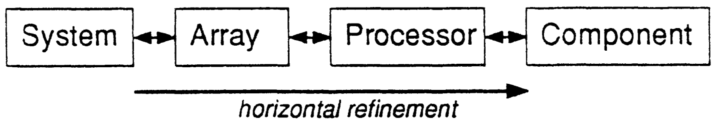


Fig. 2. A simplified design flow within the level of signal flow graphs, horizontal hierarchy

system. This may be done e.g. in defining fixed points of the desired map. At this level, there is no corresponding algorithm defined. The *behavioral specification* contains more informations on the implementation of the desired function. Consequently, the algorithm design phase is concerned with problems like numerical robustness and design of an inherent parallel algorithm which may fit to the architecture in mind. The task of the subsequent parallelization is to increase the degree of parallelism, e.g. by representing the specification in a single assignment form. As a result, parallelism in the algorithm is explicit and the partial order between the corresponding operations can be represented in form of a *dependence graph*. Architecture synthesis is concerned with the assignment of operations to computational cells, with the scheduling of these operations, and with the specification of the cells. This structural information can be represented in form of a hierarchical *data flow or signal flow graph*. At this stage, the (partial) specification of the desired architecture enters the design flow. After that, *high-level synthesis* is concerned with designing control paths, data paths and with logic synthesis. If a realization as an integrated circuit is aimed for, a *layout* generation phase must be added. The management of an iterative design flow, the consistency between the levels of abstraction, the simulation and verification between and within the levels are tasks of the design environment. The tools mainly are concerned with one level or with conversions to neighbored levels.

The purpose of the following Fig.2 is the representation of the design flow within one level of abstraction (*horizontal hierarchy*). As an example, the level of signal flow graphs is chosen. The signal flow graph corresponding to the system level contains the major subsystems of the architecture, their input and output ports and communication channels. The next level of refinement contains a more detailed representation of these pieces, which may be processor arrays. These arrays may consist of single processing elements whose signal flow graph (including terminals) is specified in the next level. Moreover, it may be desirable to refine the data types also, starting from a symbolic form, adding more and more information such as e.g. floating point representation and bit level representation. An arbitrary number of intermediate refinements is possible. The hierarchical structure may be created by the designer while specifying the systems behavior. The tools to be described in Section 5 enable a processing of this structure, i.e. flatten the hierarchy or create additional levels by partitioning. Moreover, the tools relate the specification within a certain level of abstraction to the next or previous level. As there are different interpretations of hierarchy in different levels, processing hierarchy is one of the

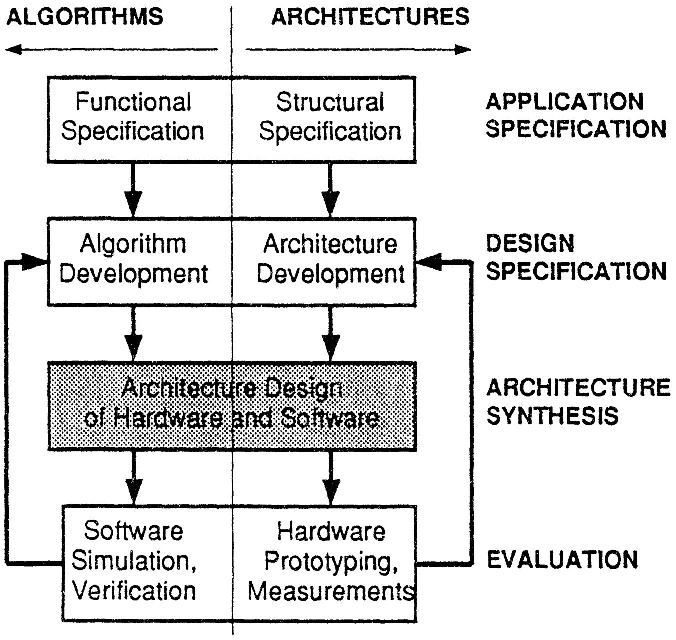


Fig. 3. Global view of the design trajectory

most important design transformations. Consequently, tools must support this strategy of stepwise refinement of specifications (behavior and structure) thereby guaranteeing a provably correct design. Obviously, this form of horizontal hierarchy is related to the concept of modules, e.g. subroutines, in programming languages (refinement of functions and refinement of data types).

The last view of a design trajectory we are going to consider is the *global view* as shown in Fig.3. Here, the iterative design trajectory through all hierarchical levels is emphasized. The structural specification restricts the final architecture, e.g. by giving a maximal number of processing elements, power and/or area limitations and I/O constraints. The step towards the design specification explores the algorithm and architecture space. Therefore, (numerically robust) algorithms which satisfy the required I/O map and fit the architecture in mind (inherent parallelism, regular data flow) must be developed and described in a behavioral way. In a similar way, as a result of the architecture development a partial specification of the final architecture is obtained. This specification is used to guide the subsequent architecture design phase and the corresponding design decisions. The tools presented in this paper are mainly concerned with the architecture design phase. Finally, in the evaluation domain, the result of the design is evaluated. The performance measures and properties are used to initialize the next iteration.

3.2. Requirements

The main responsibilities of the environment are the management of the design data and the documentation of a design, the embedding of the tools and the user interface. In the following, these tasks are visited individually:

Tool Management: Usually, a design process covers various levels of abstraction. This is due to the hierarchical design flow. In the course of a design, more and more detailed informations about the specification of a design are contributed by the user. At the same time, the design system generates the corresponding specifications and informations about the generated architecture. Referring to the above described design trajectory, e.g. the following views of a design are involved: sequential algorithm specification in an imperative language, parallel specification in a functional form, representations of dependencies in a data dependency graph, data flow or signal flow representation. If the design is refined to the level of VLSI layout, more levels are necessary, like circuit levels and layout. The tools deal with restricted views of the design data as each tool individually only solves a portion of the whole design problem. Consequently, the environment must guarantee the consistency between the various levels of refinement and must contribute interfaces between the corresponding tools. This consistency is one requirement to generate provably correct or at least verifiable designs. Moreover, because of the complexity of the design trajectory, it is necessary to manage the design flow. Therefore, informations about current design activities, missing steps in the design trajectory and a set of applicable tools at the current stage of the design must be available. These meta data about the design should be offered to the designer in order to guide his design decisions.

User Interface: Because of the complexity of the design problem we need as well the possibility for fast prototyping as a close interfacing between the designer and the design environment. The designer must be supported in bringing his own skills and experiences in the development of the current architecture.

Data Management: One of the main tasks of a design environment for the class of architectures described above is to storage and manage the design data. This task is closely related to the above described tool management. The following terms characterize some of the main requirements of the data management system: consistent management of design data, support of a gradual refinement of a design, version mechanism, concurrent data access by several designers working at the same project.

Finally, the implementation of the environment must lead to an efficient and fast handling of design data and communication between various tools at various levels of abstraction.

In the Section 4.2, some design systems for parallel signal/image processing architectures will be described which (partly) take the above mentioned criteria into account.

4. Tools

The tools which are embedded in a design environment reflect the mathematical models and concepts on which the design methodology is based. In the following we concentrate on the class of tools responsible for the transition between a given input specification (behavior) and the specification of the designed architecture including hardware and software (structure). Usually, the representations are based on a certain formal language in order to cope with verification and simulation. We do not attempt to include tools for solving problems on lower levels of abstraction, e.g. datapath synthesis, controller synthesis, microcode optimization, logic synthesis, placement, routing.

4.1. Requirements

At first, some general requirements for the tools are listed.

Embedding and Languages: The tools must fit in the whole design environment. The design language usually covers several levels of abstraction in order to avoid an insufficient communication between the tools. The (mathematical) formal languages that are used to formulate the design steps and to prove their correctness depend on the levels of abstractions the tools are dealing with. They are usually completely different form the (broad range) design language used in the environment. Because of these tasks, languages of the tools are more restricted as design languages, i.e. they represent special views of the design language. Examples of languages used for this purpose are ALPHA [10], OCCAM [11, 12], UNITY [13, 14] and Crystal [15]. The interfaces between tools and environment must guarantee a consistent relation between both kinds of languages.

Verification and Simulation: There must be the possibility to verify the design at any stage of the design process. There are mainly two possibilities to achieve this goal: simulation and verification. Whereas simulation compares input/output maps for certain sets of stimuli, verification is concerned with proving that a (sub)system obeys the corresponding specification. Both techniques must be combined as well on the tool as on the environment level. Moreover, the mathematical methods and models used should fit into a provably correct design trajectory, at least within the scope of one level of abstraction.

Scope: Following the remarks in Section 2 concerning the target algorithms, architectures and applications, the scope of the tools should cover also unavoidable deviations from regularity assumptions. These irregularities inevitable occur if array boundaries, external memory access, control flow generation and special boundary processor are taken into account in the design, i.e. are part of the final specification, see e.g. [4, 5, 16, 17].

Stepwise Refinement: There are two different aspects of the principle of stepwise refinement.

At first, the tools should support gradual horizontal refinement of a specification. For example, the designer of a system for computer graphics may at first specify the fact, that his system consists of a pipeline of processors assuming that the processors are already defined. Then these 'auxiliary' subsystems are defined assuming that some components are already defined. The designer may proceed in this manner until he reaches e.g. a fine grain specification of the system. Note, that this stepwise refinement not only concerns the behavioral specification but also the specification of the architecture as provided by the design system.

Secondly, the given input specification should be transformed to a specification of the architecture gradually by adding more and more constructs which specify the structure of the realization and which guarantee that the implementation constraints are satisfied. This vertical stepwise refinement approach enables the designer to interact with the design system at any stage and to influence the design by his own skills.

Parameterization: The use and generation of parameterized subsystems should be possible in order to shorten design times and to incorporate previous experience into a current design. Parameterization may concern the problem size, e.g. the number of input bits of a regular array multiplier. Moreover, also the specifications are required to be parameterized. Consequently, size parameters must be kept symbolic as far as possible and tools must deal with reduced quantified representations of the specification. Examples of design systems which support generic designs are HiFi [18] and Crystal [15].

Main tasks assigned to the tools are the assignment of functions to certain processing elements (assignment) and to certain times (scheduling). Of course, the performance of the final architecture depends on factors like efficiency, degree of load balancing, relation between I/O rate and computing power, number and complexity of processing elements, fault tolerance and many others. The following list contains some of the more involved basic tools which help to construct an architecture by taking some of the above factors into account.

Partitioning Usually, regular architectures such as systolic arrays or wavefront arrays are designed to solve problems of a fixed size. These full-size arrays can be extended in a modular way when the size of the problems grows. However, the size of the hardware is limited by the available resources. The problem of designing problem-size independent arrays is called partitioning in the following. There are two main classes of partitioning problems: lack of dimensions and lack of processors. In the first case, the spatial dimension of the designed architecture does not fit given constraints. For example, instead of realizing a two-dimensional processor array it is sometimes advantageous to deal with a linear array only because of the trade-off between local memory, I/O bandwidth and computation rate. There are many results concerning the solution of these problems, see e.g. [19, 20, 21, 22]. The second problem is concerned with designing architectures with limited, fixed number of processing elements, see e.g. [23, 24, 25, 26, 27, 28]. A solution to the above mentioned problems should satisfy the following requirements:

- A unified approach to the solution of the partitioning problem should be able to realize all known partitioning schemes like multiprojection (fit given dimension of architecture), active clustering (fit given number of processing elements) and passive clustering (increase efficiency of architecture).
- The partitioning must also provide for a systematic generation of control structures, i.e. the generation of control signals and the specification of control signal paths and control processors.
- The partitioning must fit into a hierarchical design trajectory based on the principles of stepwise refinement. Therefore, partitioning must be compatible with other tools like localization and control generation.
- The following factors must be taken into account: (1) minimal overall computation time, (2) minimal control overhead and (3) balanced trade-off between external communication and local memory.

An approach which combines these issues is given in [29].

Control Generation While going from behavioral specification to a structure, constructs must be added to the corresponding programs that account for the control of processing elements. In particular, processing elements may execute different operations at different time instances. Again, there are requirements an efficient control scheme must satisfy:

- Control signals must be handled equivalently to data signals in order to fit into a homogeneous design trajectory. Consequently, a regular and local flow of control signals is desired.
- The generation of control signals and the corresponding specification of control circuits within the processing elements must be done automatically.
- Processors generating the control signals (usually at the border of the array) must be specified.
- There should be the possibility to optimize as well the complexity of the control circuits within the processing elements as the complexity of the control signal generation as the interconnection paths for these signals.

Results on automatic generation of control can be found in e.g. [30, 17].

Localization The class of piecewise regular algorithms as used for the input specification is restricted not only by static parallelism but also by a regular connection of regular dependency structures. Therefore, it would be desirable to include specifications which have non-uniform and broadcast dependencies. The task of localization is to convert these algorithms into algorithms with local and uniform dependencies. As a result, a considerably more general input/output specification of the architecture can be included in the design trajectory, see e.g [31, 32, 33, 34]. In addition to these applications of a localization tool, the following problems can be addressed:

- Perform a local distribution of input data from terminals to processing elements and the distribution of output data to corresponding terminals.

- Perform a local distribution of control signals from either input terminals or special control processors. Consequently, the localization can be used for the generation of local control signals.
- Consideration of constraints concerning the interconnection structure of the designed architecture (e.g. lack of communication channels, re-routing).

Parallelization Very often, an input specification is given as an imperative program formulation containing nested loops (Fortran do-loops or C for-loops). The main reason why we consider these loop programs is that there are algorithms of a wide spectrum of applications available in this form and nested loop programs are important sources of parallelism in numerical computation programs. In practice, such programs are easy to test and modify since they can be directly compiled and executed on a sequential computer. Obviously, the data or operation dependencies are distributed among the prescribed execution ordering imposed by the nesting statements and the implicit dependencies imposed by the loop body. In order to extract the parallelism of such an apparently completely sequential formulation a detailed analysis of the dependencies among the statements is necessary. Consequently, the aim is to construct an equivalent program in an equational form. This conversion to a single assignment form enables a representation by a dependence graph. Despite of the fact that many publications have been devoted to this subject, see e.g. [35, 36], an implementable procedure valid for a broad class of loop programs has not been derived yet.

The following requirements should be taken into account:

- The tool should be applicable to a broad class of loop programs.
- As a result, an algorithm in an equational form is expected.
- The tool should accept and process parameterized specifications such that no unfolding is necessary. Otherwise, the fact that the complexity of the parallelization is dependent on the problem size prohibits the use of the tool for large size applications.

There are different solutions to the above problems known which are based on different mathematical models and methods. Some of these approaches will be reviewed in the next section.

4.2. Methodolgies and Systems

At first, some systems devoted to the design of parallel architectures for signal and image processing systems are described. From the many existing approaches, e.g. Diastol [37, 38], VACS [39], SDEF [40], Advis [41], Presage [42],COMPAR [43], Cathedral [1], HiFi [18, 44], ALPHA [10] and Crystal [15], only the last four approaches will be covered. COMPAR will be covered in Section 5.

The design system *Cathedral*, see e.g. [1], does not fit completely into the framework of this paper. This high-level architectural synthesis system is devoted to regular designs only partly. It is planned, that five different methodologies are embedded in this system each addressing a specific target domain for architectures and algorithms. The system is based on the broad range, pure functional language SILAGE. This single assignment language can be used for behavioral and structural specifications. In particular, Cathedral IV [45] is targeted at application specific regular array design. The underlying implementations and design methodologies use the Cathedral I,II and III tools for the design at low levels of abstraction. The embedded design transformations are based on the classical approach of Moldovan and Rao. Recently, some piecewise regular extensions have been adopted, see [45].

The *HiFi* system [44, 18] addresses the implementation of the design environment and the set of tools following the requirements given above. It provides an environment for the specification and design of VLSI implementable synchronous and asynchronous processor networks. The physical low levels of the design trajectory are taken care of by the CAD framework Nelsis. As in HiFi, various powerful tools are integrated around an object-oriented design data base. The basic principles of HiFi can be described as follows:

- The design language is an object-oriented implementation of the AST (applicative state transition) model. This language provides the hierarchical specification of behavior, a timing model and enables the hierarchical specification of asynchronous and synchronous processor networks. The semantics of the AST language is based on a combination of classical finite state machine descriptions and functional programming. Consequently, the main concepts are state (in extension of a pure functional approach), function and function control (as borrowed from state machine descriptions).
- The tools and specifications are fully parameterized in order to support a generic design strategy. Because of the functional aspects of the AST model, parameter instantiation is based on the λ-calculus concept. Pattern matching enables the derivation and passing of parameters.
- As well the implementation, the data management system and the design language are object-oriented. Consequently, in contrary to other approaches, the functional aspects of the design language and the transformations is simulated by a proper implementation.

The system contains tools for simulation, graphics, user interface and interactive tools for the design of regular and irregular architectures. Within the development of the design system, many theoretical results concerning the mathematics of architectural design have been published, e.g. in the areas of loop parallelization and partitioning [36, 46].

The next two systems to be described, *ALPHA DU CENTAUR* [10] and *Crystal* [15] are missing the environment. Consequently, they cover only a few levels of abstraction. On the other hand, the languages and implementation chosen are more adapted to the mathematical models and methods of the architectural design phase. In *ALPHA DU CENTAUR* [10], the design language is the functional equational language ALPHA. It is adapted to the notation of recurrence equations and describes them in a global functional way. Consequently, the semantics of this language is overloaded as it can be interpreted as a behavioral and structural specification. The implemented system is based on program rewriting as during the design process the initial program is adapted to the final architecture in a provably correct way. The transformations can be done interactively.

The approach taken in *Crystal* [15] is similar. Again, the system is based on a overloaded functional language and the principle of stepwise refinement is used. The language Crystal is based on the λ-calculus and consequently, fully parameterized. Moreover it inherited many properties from standard functional languages such as strong typing and pattern matching facilities. In extension to *ALPHA DU CENTAUR*, a meta-language has been defined which is designed to provide the designer with a formal tool to implemented transformation schemes. Meta-Crystal enables the construction and manipulation of pieces of crystal text.

Finally, some remarks on the different mathematical approaches to the design of (piecewise) regular architectures are given. As it is not the purpose to give a complete list of theoretical results in this area, some representative results are described. Moreover, detailed informations on results concerning specific tools, like partitioning, localization and control generation, are contained in the next section.

The potential of VLSI for massive parallel computations was first recognized by Kung and Leiserson [2]. After they introduced the term *systolic array* for regular mesh connected synchronous processor arrays many algorithms from areas like signal processing, image sequence processing, linear algebra and combinatorial optimization have been mapped onto this computer architecture. About the same time, synthesis methods for VLSI processor arrays received much attention. In principle, behavioral descriptions are transformed in functions that distribute operations over time and space. The first pioneering results have been obtained by Kuhn [47] who used linear index transformations for relating algorithms and their systolic implementation. This result has been recovered later by Moldovan [8]. The extensions of Quinton [48], Miranker et al. [49] and Capello et al. [50] to this work resulted in the theory of *regular iterative algorithms* as proposed by Rao [9, 51]. He linked results obtained so far to the work of Karp, Miller and Winograd [52] on *uniform recurrence equations* and he introduced considerable extensions to the synthesis methodology. These methods are used increasingly to parallelize loop programs for massive parallel architectures, see e.g. [53, 54].

The approaches mentioned so far have a simple geometrical interpretation: The given single assignment algorithm is represented by a *dependence graph*. The dependence graph represents the available parallelism of the given algorithm as it determines the partial order of operations. The mapping of *uniform recurrence equations* onto processor arrays corresponds to an affine transformation of the index space and a subsequent interpretation of the coordinates as time and space.

Recently, methods which aim at mechanical, provably correct synthesis methods are receiving more and more attention, see e.g. [14, 12, 15]. They are based on *formal program transformations* and use concepts like functional programming languages, λ-calculus and strong typing of functions.

The approach proposed in the next section combines as well the geometric interpretation as the stepwise transformation of specifications. The former serves to design (in a sense) optimal architectures whereas the latter is responsible for the certifiable correctness of the design.

5. A Piecewise Regular Design Methodology

The main subject of this chapter is a methodology which serves to fullfil the requirements given in Section 4.1. It is based on experiences with the system COMPAR (COmpiler for Massive Parallel ARchitectures), see e.g. [43]. At first, the main characteristics of the proposed approach to the architecture synthesis and some aspects of a software implementation are given. Then, the mathematical models and methods on which the design trajectory is based are introduced.

5.1. Implementation

The implementation of COMPAR is based on the following main principles:

Mathematical models and methods The class of algorithms which can be efficiently mapped on processor arrays (*piecewise linear algorithms*) has been carefully chosen. On the one hand, it is general enough such that most algorithms of the target applications are covered. On the other hand, parameterized tools which solve the design problem can be constructed. Even after applying transformations like mixing hierarchical levels, partitioning, scheduling, the resulting architectural specification is a *piecewise linear program* which can be processed further. As a result, a homogeneous design trajectory is obtained. The main mathematical methods used are *operations on index sets*. The fact, that the required operations, like cutset, convex hull, affine mapping, can be carried on parameterized index sets is responsible for the efficiency of the tools, e.g. no enumeration of operations or index sets at compile time is necessary.

Programs-Algorithms-Architectures The semantics of a piecewise regular/linear program is 'overloaded', i.e. the program can be interpreted as a behavioral description in form of an algorithm and as a structural description as an architecture. The programs are formulated using a notation similar to the UNITY [13] program scheme. This programming notation has been developed especially for the design of parallel programs using the concept of stepwise refinement of program specifications. In particular, we are using the equational scheme, a functional subset of UNITY. The interpretation of a piecewise linear program as a specification of hardware considers also the control of processor functions and interconnection paths that are automatically generated as part of the design trajectory.

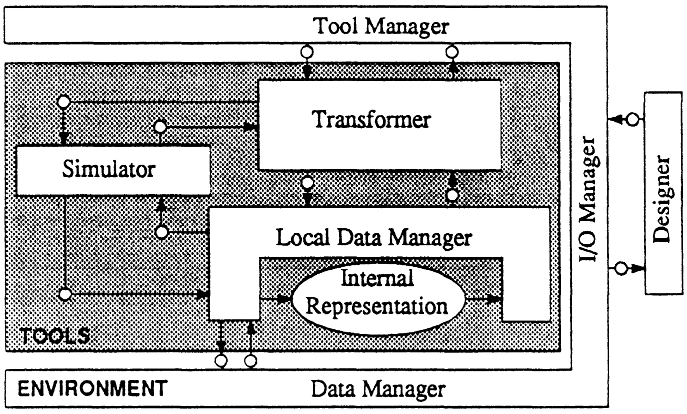


Fig. 4. Software model of the COMPAR design system

Transformative Approach All the above mentioned properties lead to a parameterized stepwise refinement approach to the design problem. The overall process of mapping consists of a sequence of program transformations that is applied to an initial program. These stepwise refining program transformations gradually change a given program until it meets the given architectural specifications. In order to guarantee a correct design trajectory, any program transformation must be semantic preserving. To this end, the tools like control generation and partitioning are further refined into basic transformations which can simply be proven to be correct. As a result, we are lead to a certifiable correct homogeneous design flow.

The software implementation of the COMPAR system is described now. The main software components are displayed in Fig.4. In the following, the main parts of the system are described in detail. Here, we concentrate on the tools as their implementation is closely related to the underlying methods and models.

- The *I/O-Manager* dispatches designs. A design is given by a program that represents a *piecewise linear algorithm.* The program is parsed and converted into an internal representation. During the design session, program transformations will be applied to this internal representation. Apart from opening and saving designs, the *I/O-manager* consists of backend generators for different output formats and target architectures. For example, it is possible to create an OCCAM program that permits the simulation of a design on a network of transputers. Also generators connected to commercial VLSI design systems are implemented.
- After creating the internal representation of the given program, the user may activate the *Graphical and Numerical Simulator.* The results of the evaluation phase may influence further design steps.

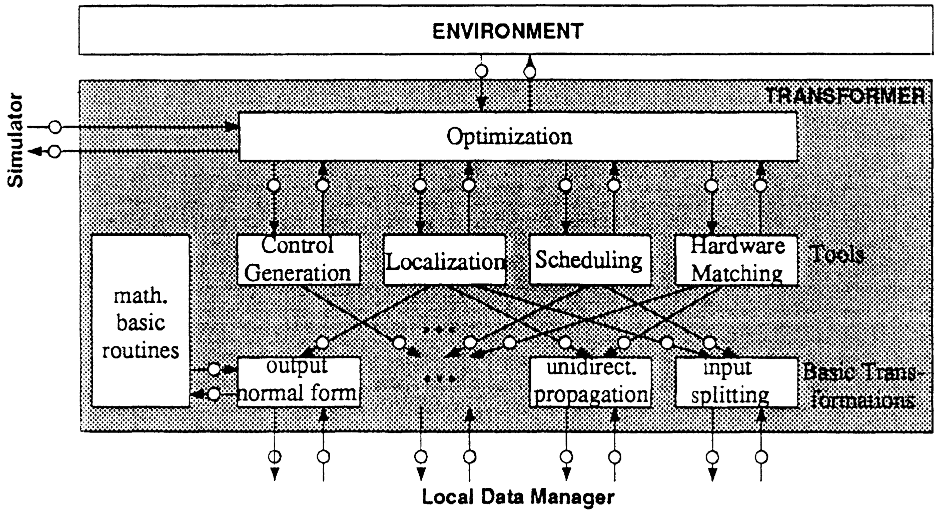


Fig. 5. Refinement of the transformer in Fig.4

– While the instances described until now basically represent the environment of the system, the *Transformer* is responsible for the proper process of mapping. The optimization unit selects a tool or a sequence of tools that convert the program into an input/output equivalent program. Each tool calls and controls a number of parameterized basic program transformations (see Fig.5) that represent entities that can be proven correct. However, the software hierarchy may be further refined to the level of mathematical basic routines

This shell model makes it possible to develop levels of software (e.g. basic transformation or tools level) independently from other levels. The consequences resulting from modifications of data structures may be limited (information hiding). Also the access of the *Transformer* to the internal representation of the design is limited to an encapsulated software instance called *Local Data Manager*. This shell structure of the software is ideally supported by the technique of object oriented programming and in particular by the programming language C++ that has been chosen as the programming language for the implementation of the COMPAR system.

5.2. Program-Behavior-Structure

The purpose of this chapter is to introduce the class of algorithms we are considering called *piecewise linear* or *piecewise regular algorithms* and their representation by a class of programs called *piecewise linear/regular programs*. In addition to [55, 6, 34], hierarchical constructs are added to the specification of the notations. It is shown that such a program can be directly related to the specification of

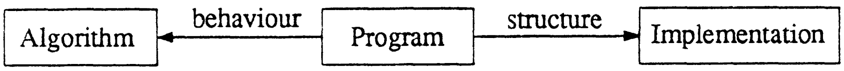


Fig. 6. Relation between an algorithm, a program, and its implementation

a *piecewise linear/regular processor array* including the hierarchical specification of processing elements, interconnection paths and the distribution of register elements. This relation between algorithms, programs and processor array is shown in Fig.6.

5.2.1. *Program Notation*

In the following the notation introduced by Chandy and Misra [13] is adopted. In particular, a functional subset of UNITY is used. As we are not going to develop a programming language, e.g. type declarations are not defined. Instead, the main purpose is to define a formal notation which describes the mathematical models underlying the piecewise regular design strategy.

An *equation* of the form $x = y$ defines that the variable x has the value of y. The sign $\|$ is used in order to separate equations. For example, $y = z \| x = y$ defines that x and y have the value of z. Note, that the ordering of equations is arbitrary. Moreover, we define that $(a, b) = (c, d)$ is equivalent to $a = c \| b = d$. The term (a, b) is called a *tupel* of a and b.

The notation of a *quantified equation* can be used to describe sets of equations easily. The *quantified equation* $\langle \| I : I \in \mathbf{I} :: S[I] \rangle$ where $\mathbf{I} \subset \mathbf{Z}^s$ is an *index space*, $I \in \mathbf{Z}^s$ is an *index vector* and each $S[I]$ is an equation denotes an enumeration of equations. Of course, $S[I]$ may also be a set of equations or even a set of quantified equations. For example, $\langle \| i, j : i = 1 \wedge 1 \leq j \leq 2 :: x[i, j] = x[0, j] \rangle$ is equivalent to $x[1, 1] = x[0, 1] \| x[1, 2] = x[0, 2]$.

Now, the notation for a *function* can be introduced. At first, $x = f(y, z)$ defines that x has the value of $f(y, z)$ where f is a function with the two arguments y and z. Following the above notation, $(a, b) = f(c, d)$ denotes that f defines both, a and b.

The notation for *conditional equations* can be seen using the following equation which defines that x has the absolute value of y:

$$x = y \text{ if } y \geq 0 \sim -y \text{ if } y < 0$$

The *cases* are separated by the symbol $\sim$. The expression $y < 0$ is called the *conditional* for the case $x = -y$. If the conditional is of the form $S[I]$ if $I \in \mathbf{I}_c$, then $\mathbf{I}_c$ is called *condition space* of the corresponding case.

Before we proceed, a simple example serves to explain the notations introduced so far.

EXAMPLE 1. *The following example describes an algorithm for the matrix-vector product $C = AB$. The* module *has the form:*

```
**mavec1**
  in
    $(\langle \| i,j : 1 \le i,j \le n :: a[i,j] \rangle,$
    $\langle \| i : 1 \le i \le n :: b[i] \rangle)$
  always
    $\langle \| i,j : 1 \le i,j \le n ::$
      $c[i,j] = 0 \quad \text{if} \quad j = 1$
              $\sim c[i,j-1] + a[i,j]b[j] \quad \text{if} \quad j > 1 \rangle$
  out
    $\langle \| i : 1 \le i \le n :: c[i,n] \rangle$
```

In addition to the above given notation, we have included a *module* name ('mavec1' in the example) and a notation for defining input and output variables. They are given by tupels of quantified variables.

It is useful to apply functions not only to variables but also to sets of variables. To this end, the following notation is used: The expression $f(\langle \| I : I \in \mathbf{I} :: a[I] \rangle)$ denotes that f is applied to a set of variables which contains $a[I]$ for all $I \in \mathbf{I}$. Equivalently, the result of a function can define a tupel of quantified variables.

EXAMPLE 2. *The example describes the same algorithm as given above. Now, the inner product is hidden in a module called 'inpro'. The two modules are defined as follows:*

```
**mavec2**
  in
    $(\langle \| i,j : 1 \le i,j \le n :: a[i,j] \rangle,$
    $\langle \| i : 1 \le i \le n :: b[i] \rangle)$
  always
    $\langle \| i : 1 \le i \le n ::$
      $c[i] = inpro(\langle \| j : 1 \le j \le n :: a[i,j] \rangle, \langle \| j : 1 \le j \le n :: b[j] \rangle)$
  out
    $\langle \| i : 1 \le i \le n :: c[i] \rangle$

**inpro**
  in
    $\langle \| j : 1 \le j \le n :: \hat{a}[j] \rangle,$
    $\langle \| j : 1 \le j \le n :: \hat{b}[j] \rangle$
  always
    $\langle \| j : 1 \le j \le n ::$
      $\hat{c}[j] = 0 \quad \text{if} \quad j = 1$
            $\sim \hat{c}[j-1] + \hat{a}[j]\hat{b}[j] \quad \text{if} \quad j > 1 \rangle$
  out
    $\hat{c}[n]$
```

The definition of a module as given above needs some more explanations:

- No recursive module definitions are allowed, e.g. a module must not need its own definition.
- All variables are local to a module. Therefore, in the preceeding example, it would be possible to use a, b and c instead of $\hat{a}, \hat{b}$ and $\hat{c}$, respectively.
- Each module defines a (global) function whose identifier is the name of the module.
- The mechanism of applying a function to a tupel of quantified variables is more involved. Let us suppose that a function f is applied to a quantified variable as follows: $b = f(\langle \| \; I : I \in \mathbf{I} :: \; a[g(I)] \rangle)$. Here, g is an arbitrary integer function which maps $I \in \mathbf{I}$ to the index $g(I)$ of the variable a. We suppose that $g(I)$ is one-to-one for $I \in \mathbf{I}$. For example, if $I = \begin{pmatrix} i \\ j \end{pmatrix}$, $\mathbf{I} = \{i, j \;:\; 1 \leq i, j \leq n\}$ and $g(I) = i + nj$, then we have $b = f(\langle \| \; i, j : 1 \leq i, j \leq n :: \; a[i + nj] \rangle)$.
  Now, the input definition of the module defining f must contain a quantified local variable. *Moreover, it is required that the definition of the local variable's index space is equal to that of the quantified variable to which the function is applied.* In the example, the input of the module defining f may contain a quantified variable of the form $\langle \| \; I : I \in \mathbf{I} :: \; \hat{a}[h(I)] \rangle$.
  This constraint is based on the fact that we interpret the relation between function application and module definition as a quantified equation. In the example, we have $\langle \| \; I : I \in \mathbf{I} :: \; \hat{a}[h(I)] = a[g(I)] \rangle$. It is not specified how the variables and the (parameterized) index spaces are passed from the function application to the module. For example, principles known from functional languages, e.g. λ-calculus, may be used.

5.2.2. Index Spaces

Obviously, index spaces are of importance in the previously introduced program notation. Moreover, it will be shown that most of the mathematical methods used in the design trajectory are based on operations on index sets. Examples of these operations are

- convex hull of index spaces,
- cutset of index spaces,
- projection of index spaces, and
- affine transformation of index spaces.

Moreover, in order to transform programs without enumeration of index spaces, and to process parameterized index spaces we restrict ourselves to the class of *linearly bounded lattices.*

DEFINITION 1. *A linearly bounded lattice is an index space of the form*

$$\mathbf{I} = \{I : \; I = A\kappa + b \;\wedge\; C\kappa \geq d \;\wedge\; \kappa \in \mathbf{Z}^{l}\}$$

where $A \in \mathbf{Z}^{s \times l}$, $b \in \mathbf{Z}^{s}$, $C \in \mathbf{Z}^{m \times l}$ *and* $d \in \mathbf{Z}^{m}$.

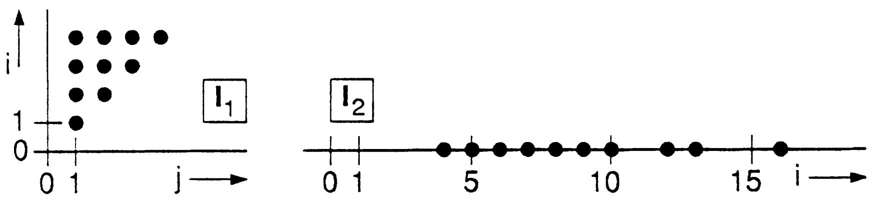


Fig. 7. Examples of linearly bounded lattices

Obviously, $\{\kappa : C\kappa \geq d \ \wedge \ \kappa \in \mathbf{Z}^l\}$ defines the set of all integer vectors within a polytope. The polytop is characterized by a set of linear inequalities. This set is mapped on I using an affine function, e.g. $I = A\kappa + b$.

It can be shown, that the following property holds:

PROPERTY 1. *The set of linearly bounded lattices is closed under the operations convex hull of union, cutset, projection and affine transformation.*

The following example serves to clarify the above notations.

EXAMPLE 3. *Let us suppose that an index set* I_1 *is defined by* $A = E$, *where* E *denotes the unity matrix, and* $b = 0$. *Then we have* $\mathrm{I}_1 = \{I : I = \kappa \ \wedge \ C\kappa \geq d \ \wedge \ \kappa \in \mathbf{Z}^l\}$, *or equivalently,* $\mathrm{I}_1 = \{I : CI \geq d \ \wedge \ I \in \mathbf{Z}^l\}$. *For* $I = \begin{pmatrix} i \\ j \end{pmatrix}$, $C = \begin{pmatrix} 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix}$, $d = \begin{pmatrix} 1 \\ -n \\ 0 \end{pmatrix}$ *and* $n = 4$ *the index set shown on the left hand side of Fig.7 is obtained. Moreover, Fig.7 also shows the linarly bounded lattice* $\mathrm{I}_2 = \{i : i = \kappa_1 + 3\kappa_2 \ \wedge \ \kappa_2 \leq \kappa_1 \leq 4 \ \wedge \ \kappa_2 \geq 1\}$. *It is obvious, that* I *is not necessarily a lattice or a convex set.*

The implementation of the above given basic operations on index sets uses a set of basic mathematical routines as shown in Fig.5. It contains algorithms for

- linear programming, projection, Fourier-Motzkin elimination,
- elimination of redundant inequalities,
- Hermite Normal Form and Smith Normal Form computations and
- convex hull of union of lattices and polytopes.

In order to be more precise, it is shown that the set of linearly bounded lattices is closed under affine transformations and cutsets.

Affine Transformation of Linearly Bounded Lattices: As an example of operations applied to index spaces, the affine transformation is elaborated. To this end, the affine integer function

$$f(I) = \Lambda I + \gamma$$

is defined which is one-to-one for all $I \in \mathbf{I}$. Now, the affine transformed index space $\mathbf{J} = f(\mathbf{I})$ is defined by

$$I \in \mathbf{I} \Leftrightarrow f(I) \in \mathbf{J}$$

With $\mathbf{I} = \{I : \ I = A\kappa + b \ \wedge \ C\kappa \geq d \ \wedge \ \kappa \in \mathbf{Z}^l\}$ we directly obtain

$$f(\mathbf{I}) = \{J : \ J = \Lambda A\kappa + (\Lambda b + \gamma) \ \wedge \ C\kappa \geq d \ \wedge \ \kappa \in \mathbf{Z}^l\}$$

EXAMPLE 4. *$\mathbf{I}_1$ of Example 3 is transformed using $\lambda = \begin{pmatrix} 1 & n-1 \end{pmatrix}$ and $\gamma = 0$. We obtain*

$$f(\mathbf{I}_1) = \left\{ J : \ J = \begin{pmatrix} 1 & n-1 \end{pmatrix} \begin{pmatrix} \kappa_1 \\ \kappa_2 \end{pmatrix} \wedge \begin{pmatrix} 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \kappa_1 \\ \kappa_2 \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 0 \end{pmatrix} \right\}$$

For $n = 4$ the index space $\mathbf{I}_2$ of Example 3 results.

Cutset of linearly bounded lattices: The set $\mathbf{K} = \mathbf{I} \cap \mathbf{J}$ with

$$\mathbf{I} = \{I : \ I = A\kappa + b \ \wedge \ C\kappa \geq d \ \wedge \ \kappa \in \mathbf{Z}^{l_I}\}$$

$$\mathbf{J} = \{J : \ J = R\kappa + s \ \wedge \ T\kappa \geq u \ \wedge \ \kappa \in \mathbf{Z}^{l_J}\}$$

is given by

$$\begin{aligned}\mathbf{K} \ = \ \{K : \ K = A\kappa + b \ \wedge \ A\kappa + b = R\xi + s \ \wedge \ C\kappa \geq d \ \wedge \ T\xi \geq u \ \wedge \\ \kappa \in \mathbf{Z}^{l_I} \ \wedge \ \xi \in \mathbf{Z}^{l_J}\}\end{aligned}$$

Obviously, this index set is not yet in the form of a linearly bounded lattices. Therefore, we evaluate the set of vectors $\begin{pmatrix} \kappa \\ \xi \end{pmatrix}$ which satsify $A\kappa + b = R\xi + s$. The results concerning the integer solution to a set of linear equations given in the Appendix are used. If a solution exists, the set

$$\begin{pmatrix} \kappa \\ \xi \end{pmatrix} \in \left\{ \begin{pmatrix} \kappa \\ \xi \end{pmatrix} : \ \kappa = e + G\eta \ \wedge \ \xi = f + H\eta \ \wedge \ \eta \in \mathbf{Z}^r \right\}$$

is obtained for some scalar $r \in \mathbf{Z}$ and vectors and matrices e, f and G, H. Using this representation the representation of $\mathbf{K}$ in standard form is obtained:

$$\mathbf{K} = \left\{ K : \ K = AG\eta + (Ae + b) \ \wedge \begin{pmatrix} CG \\ TH \end{pmatrix} \geq \begin{pmatrix} d - Ce \\ u - Tf \end{pmatrix} \ \wedge \ \eta \in \mathbf{Z}^s \right\}$$

Similarly, it can be shown that the set of linearly bounded lattices is closed under the other mentioned basic operations.

5.2.3. Piecewise linear and piecewise regular algorithms

The class of *piecewise linear algorithms* has been defined in [33, 34]. This class extends the notion of *regular iterative algorithms* [9] that may be related to regular processor arrays such as systolic and wavefront arrays. Piecewise linear algorithms have the following properties:

- A piecewise linear algorithm consists of a set of equations that relate linearly indexed variables. For example, $x[PI+f] = \mathcal{F}(\dots, y[QI-d], \dots)$ with $I \in \mathbf{Z}^s$ and the rational matrices and vectors $P \in \mathbf{Q}^{n_x \times s}$, $Q \in \mathbf{Q}^{n_y \times s}$, $f \in \mathbf{Q}^{n_x}$ and $d \in \mathbf{Q}^{n_y}$ is such an equation. $\mathcal{F}$ denotes an arbitrary function and x, y are indexed variables.
- Equations are quantified, i.e. an equation defines the indexed variable on its left hand side, say e.g. x, for a set of values of the iteration vector I. Such a set is called *iteration space*. For example, $\langle \| \; I : \; I \in \mathbf{I} :: \; x[PI+f] = \mathcal{F}(\dots, y[QI-d], \dots) \rangle$ where $\mathbf{I}$ is an *index space* is such a quantified equation. The affine expressions $PI+f$ and $QI-d$ are called *index functions*. These index functions map $I \in \mathbf{I}$ to *integer* vectors, i.e. $PI+f \in \mathbf{Z}^{n_x}$ and $QI+d \in \mathbf{Z}^{n_y}$ for all $I \in \mathbf{I}$. *Each equation may be assigned a private iteration space.* Quantified equations can be *expanded* by copying the equation for each instance $I \in \mathbf{I}$, e.g. $\langle \| \; i : \; 0 < i \leq 2 :: \; x[i] = \mathcal{F}(\dots, y[i-1], \dots) \rangle$ results in $x[1] = \mathcal{F}(\dots, y[0], \dots) \; \| \; x[2] = \mathcal{F}(\dots, y[1], \dots)$ after expansion.
- The iteration spaces assigned to each quantified equation are *linearly bounded lattices.*
- After expansion, any instance $x[J]$, $J \in \mathbf{Z}^n$ of any indexed variable x appears at most once on the left hand side of an equation (single-assignment property) and there exists a partial ordering of the equations such that any instance of any variable appearing on the right hand side of an equation appears on the left hand side earlier in the partial ordering (computability).

A piecewise linear algorithm can be written in form of a program which consists of a set of quantified equations as follows:

$$\dots \; \| \; \langle \| I : \; I \in \mathbf{I} \; :: \; x[PI+f] = \mathcal{F}(\dots, y[QI-d], \dots) \rangle \; \| \; \dots$$

where P, Q and f, d are integer indexing matrices and vectors of appropriate dimensions, and $\mathcal{F}$ is an arbitrary function.

EXAMPLE 5. *A simple example of a piecewise linear algorithm with just one quantification is the algorithm of a finite impulse reponse (FIR) digital filter:*

```
FIR
  in
    $(\langle \| j:\ 0 \le j < N :: a[j]\rangle, \langle \| i:\ 0 \le i < T :: u[i]\rangle)$
  always
    $\langle \| i,j:\ 1 \le j < N \wedge N-1 \le i < T :: y[i,j] =$
        $y[i,j-1] + a[j]u[i-j]\rangle \|$
    $\langle \| i:\ 0 \le i :: y[i,0] = a[0]u[i]\rangle$
  out
    $\langle \| i:\ N-1 \le i < T :: y[i,N-1]\rangle$
```

For the first quantification in the always section we have $P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $Q_y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $Q_a = (0\ 1)$, $Q_b = (1\ -1)$, $f = (0\ 0)^t$, $d_a = d_u = 0$ *and* $d_y = (0\ 1)^t$. *Another piecewise linear algorithm is that given in Example 1. The always section of 'mavec1' can be equivalently written as*

$$\langle \| i,j : 1 \le i \le n \ :: \ c[i,1] = 0\rangle \ \|$$
$$\langle \| i,j : 1 \le i \le n \ \wedge \ 2 \le j \le n \ :: \ c[i,j] = c[i,j-1] + a[i,j]b[j]\rangle$$

Using the fact that the cutset of linearly bounded lattices is again a linearly bounded lattice it can be shown, that the following property holds:

PROPERTY 2. *Any module*

- *which is correct according to our notation,*
- *whose index spaces and condition spaces are linearly bounded lattices,*
- *whose indexing functions are all affine, and*
- *which satisfies the computability and single assignment properties after expansion*

can be interpreted as a piecewise linear algorithm.

Programs that satisfy these properties are called *piecewise linear programs.* We will show later, that this property also holds for a hierarchical set of modules, see also Examples 1 and 2. In the design of piecewise linear processor arrays, to a given program arbitrary transformations can be applied which preserve the I/O behavior and which lead to piecewise linear programs. Some useful program transformations will be explained in Section 5.

The dependency structure of a piecewise linear algorithm can be represented by a *dependence graph* which can be constructed as follows:

1. The given program must be expanded such that a set of equations is obtained.
2. To each instance of an indexed variable there is associated a vertex of the dependence graph.
3. If an indexed variable $x[I]$ directly depends on an indexed variable $y[J]$, then there is an edge from the vertex associated to $y[J]$ to that associated to $x[I]$.

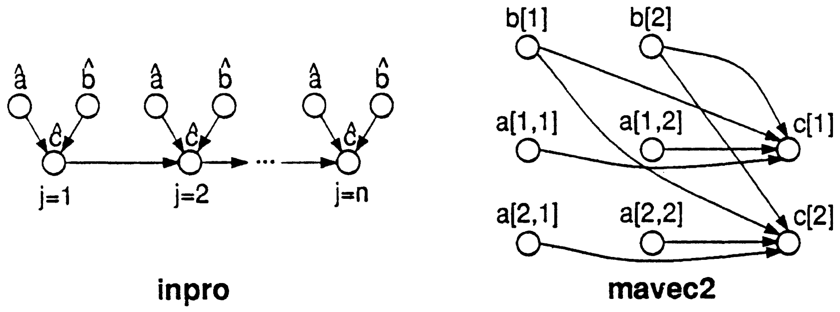


Fig. 8. Dependence graph corresponding to modules inpro and mavec2

Very often, we will also give a geometrical representation of a dependence graph. In particular, this is possible if all variables have the same number of index dimensions. A vertex corresponding to a variable $x[I]$ for some instance of I is placed at the index point I. The dependence graph represents a partial ordering of the functions and therefore, represents the degree of parallelism available in the algorithm.

EXAMPLE 6. *The geometrical representation of the dependence graph corresponding to the module 'inpro' in Example 2 is shown in Fig.8. The same figure also contains the dependence graph corresponding to the module 'mavec2' of Example 2 for $n = 2$.*

Now, the class of *piecewise regular* algorithms can be defined, see e.g. [16, 6]. The only difference to the class of piecewise linear algorithms is the fact, that all indexing functions in the always section are constant. Consequently, the matrices P and Q which have been used for the definition of piecewise linear algorithms are unity matrices. Therefore, all variables in the algorithm are of the form $x[f(I)]$ where I is the index vector and the indexing function $f(I) = I + d$ where d is some constant integer vector of appropriate dimension. A piecewise regular algorithm can be represented as a program of the following form:

$$\ldots \parallel \langle \| I : I \in \mathrm{I} :: x[I + f] = \mathcal{F}(\ldots, y[I - d], \ldots) \rangle \parallel \ldots$$

The following property can be shown to hold:

PROPERTY 3. *Any module*

- *which is correct according to our notation,*
- *whose index spaces and condition spaces are linearly bounded lattices,*
- *whose indexing functions are all constant, and*
- *which satisfies the computability and single assignment properties after expansion*

can be interpreted as a piecewise regular algorithm.

Programs with these properties are called *piecewise regular.*

The following example gives a piecewise regular program for the FIR filter algorithm.

EXAMPLE 7. *Again, the algorithm for an FIR filter is used. After some program transformations the piecewise linear algorithm given in the previous example can be converted into the following piecewise regular algorithm:*

```
PRFIR
  in
    $(\langle \| j : 0 \leq j < N :: a'[0,j] \rangle, \langle \| i : 0 \leq i < T :: u'[i,0] \rangle)$
  always
    $\langle \| i,j : 0 \leq j < N \wedge j \leq i < T ::$
    $y[i,j] = a[i,j]u[i,j]$  if  $j = 0$
          $\sim y[i,j-1] + a[i,j]u[i,j]$  if  $j \geq 1 \;\|$
    $a[i,j] = a[i-1,j]$  if  $i \geq 1$
          $\sim a'[i,j]$  if  $i = 0 \;\|$
    $u[i,j] = u[i-1,j-1]$  if  $j \geq 1$
          $\sim u'[i,j]$  if  $j = 0$
    $\rangle$
  out
    $\langle \| i : N-1 \leq i < T :: y[i,N-1] \rangle$
```

The program 'inpro' in Example 2 is piecewise regular also, as there are only constant index dependencies, see Fig.8.

The importance of the class of piecewise regular algorithms is based on the following properties:

- If the iteration spaces corresponding to the equations are equal for all quantifications, then the class of *regular iterative algorithms* as defined by Rao [9] is obtained. Based on this notation, regular processor arrays can be synthezied. On the other hand, the class of piecewise regular algorithms leads to processor architectures which are still local. They consist of interconnected regular subsystems.
- The given specifications are often not completely regular. Consequently, they cannot be represented by a regular iterative algorithm. If they can be partitioned into regular subalgorithms, e.g. a sequence of loop constructs or programs with subroutine calls then a piecewise regular specification is possible.
- If the final specification of the architecture includes I/O of data, boundary processors and special control processors, a regular design methodology is not approporiate.
- Properties 1 and 3 indicate, that the class is closed under certain program transformations. In particular, partitioning, localization and control generation can be carried out without the necessity to extend the proposed algorithm models.

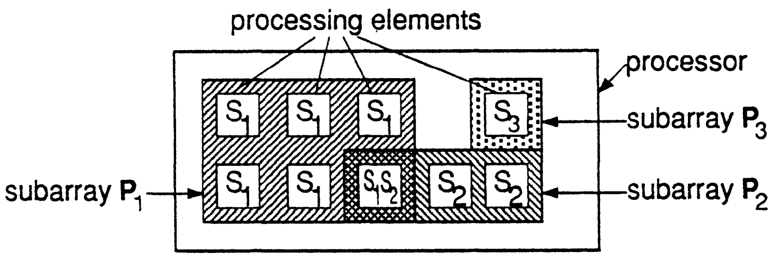


Fig. 9. Decomposition of a processor in subarrays and processing elements

5.2.4. Piecewise linear/regular processor arrays

Given a piecewise linear program as described in Property 2 or a piecewise regular program according to Property 3, the structure of a processor that executes this specification has not yet been described.

There are many possible interpretation schemes for the defined program classes. These schemes depend on the particular architecture specification, e.g. hierarchical vs. flat representation, local control vs. global control, FIFO vs. LIFO stacks, available predefined subsystems and the chosen interconnection structure. Therefore, the following scheme may be understood as an example only. The hierarchical specification of processor arrays is described for a special class of piecewise linear programs. A specialization to constant index functions can simply be carried out.

The interpretation is based on the assumption, that the given program has been transformed beforehand using program transformations which are described in the next section. This form can be described as follows:

- The program consists of a set of modules. As there are no recursive module definitions, a partial ordering of modules can be derived (hierarchy), e.g. the function defined by the module on the top level is not used by any other module.
- The always section of each module consists of a set of quantifications of the form
  $$\langle \| \, p : \; p \in \mathbf{P}_1 :: \; S_1[p] \rangle \; \| \ldots \| \; \langle \| \, p : \; p \in \mathbf{P}_V :: \; S_V[p] \rangle$$
  Each quantification specifies a *subarray* of the *processor* corresponding to the module. The spaces $\mathbf{P}_i$ are called *processor spaces*, p is a *called processor index*. *Processing elements* associated to each term $S_i[p]$ exist at all processor indices $p \in \mathbf{P}_i$. It is possible that several processing elements are overlaid at one processor index, see Fig.9.
- Each term $S_i[p]$ denotes a set of equations of the form
  $$C_1^i[p] \; \| \; C_2^i[p] \; \| \ldots \| \; C_{W^i}^i[p]$$
  These equations are interpreted as the components of a processing element corresponding to $S_i[p]$, see Fig.10. To reduce the notational complexity, let us consider the components of one processing element $S[p]$ only. The components

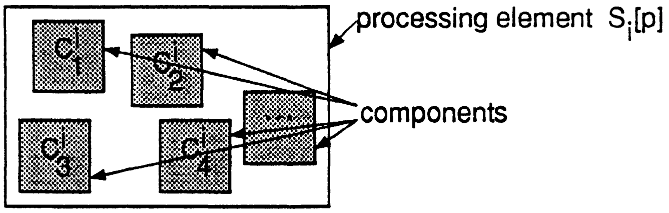


Fig. 10. Decomposition of a processing element in components

are the actual computing devices. Obviously, there is a vast number of possible program schemes and interpretations for these components.

– For example, a processing element may be composed of components like

$$\langle \| \, t: \ t \in \mathrm{T}_1[p] :: \ x[p, t+s] = \mathcal{F}_1(..., y[p-u_1, t-r_1], ...) \rangle \ \|$$

$$\langle \| \, t: \ t \in \mathrm{T}_2[p] :: \ x[p, t+s] = \mathcal{F}_2(..., y[p-u_2, t-r_2], ...) \rangle \ \| \ ...$$

Here, r_i and s are scalars, $\mathrm{T}_i[p]$ are called *sequencing spaces* which may depend on the processor index p, t is called *sequencing index* and u_i are integer vectors of appropriate dimension. Of course, components defining other variables may be also part of a processing element. Obviously, the functions $\mathcal{F}_i$ correspond to purely stateless (combinatorial) circuits with a single output.

According to Fig.11 a processing element $S[p]$ can be realized as follows: For any i there is a unidirectional link from the processing element at location $p-u_i$ whose output is $y[p-u_i, t]$. The link contains a synchronous FIFO (first in / first out) register of length r_i. There is a FIFO of length s at the output x of the processing element. The different equations defining the variable x are implemented using multiplexers. At first it is computed which condition $t \in \mathrm{T}_i[p]$ is true. The output x of the processing element is assigned to that of component i.

It can be seen that different processing elements exist due to different processor spaces. The computation of iteration dependent conditionals of the form $t \in \mathrm{T}_i[p]$ can be avoided and computed outside the processor array by control generation (see [17]). Control generation removes condition spaces by locally propagating control signals from the border of the array to each processing element. These control variables are connected to the (de)multiplexers in Fig.11. In that case, a processing element does neither need to count the sequencing index, nor its processor index must be stored. The conditionals $t \in \mathrm{T}_i[p]$ may, however, also be computed inside the processing elements (see Fig.11). As linear operations on processor and sequencing indices have to be carried out, comparators, adders and modulo counters realize these conditionals.

– Of course, a processing element may also contain components which realize a function applied to a tupel of quantified variables. Let us suppose that the components have the form

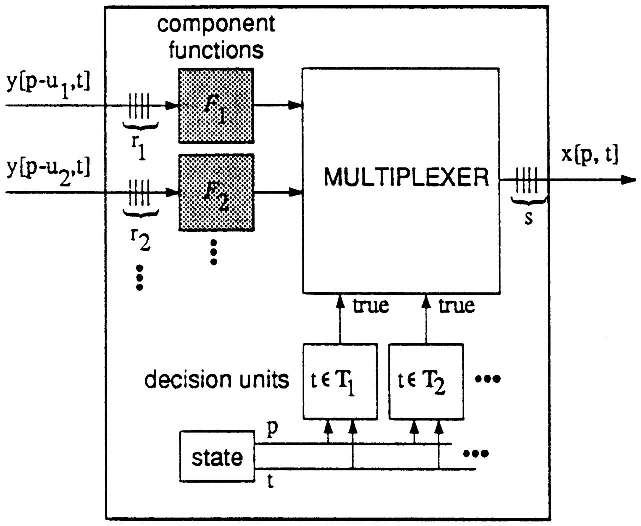


Fig. 11. Part of a processing element with stateless modules

$$\begin{aligned}&\langle \| \, t : \; t \in \mathrm{T}_1[p] :: \langle \| \, q : \; q \in \mathbf{Q}_1[p] :: x[p,q,t+s] \rangle \rangle = \\ &\quad \mathcal{F}_1(\ldots, \langle \| \, t : \; t \in \mathrm{T'}_1[p] :: \langle \| \, q : \; q \in \mathbf{Q'}_1[p] :: y[p-u_1,q,t-r_1] \rangle \rangle, \ldots) \, \| \\ &\langle \| \, t : \; t \in \mathrm{T}_2[p] :: \langle \| \, q : \; q \in \mathbf{Q}_2[p] :: x[p,q,t+s] \rangle \rangle = \\ &\quad \mathcal{F}_2(\ldots, \langle \| \, t : \; t \in \mathrm{T'}_2[p] :: \langle \| \, q : \; q \in \mathbf{Q'}_2[p] :: y[p-u_2,q,t-r_2] \rangle \rangle, \ldots) \, \| \ldots\end{aligned}$$

The interpretation is similar to that given in the previous case, see Fig.12. The main difference can be seen in the fact that bundles of signals are processed and the component functions may have internal states. The above given components define values of $x[p,q,t]$ for all $q \in \mathbf{Q} = \bigcup_{(i)} \mathbf{Q}_i$. In a similar way, bundles of signals are inputs to the components $\mathcal{F}_1, \mathcal{F}_2, \ldots$.

It may be possible also that the functions define several quantified variables. In this case, the interpretation scheme can be extended easily. Note that the components $\mathcal{F}_1, \mathcal{F}_2, \ldots$ may be defined by other modules. Moreover, similar remarks than at the end of the previous item concerning the interpretation of simple combinatorial components apply: there are many different ways to physically realize the multiplexers and decision circuits.

- Finally, Fig.13 describes the whole hierarchical structure of the proposed formal architectural description.

Note that there are also other possible interpretation schemes for piecewise linear/regular programs: For example, computing the equations and the iteration dependent conditionals may be realized in software if the processor array consists of programmable processing elements (e.g. multiprocessors). Also intermediate hardware/software interpretations of piecewise regular program schemes can be thought of. In the design system COMPAR (see [56, 43]), the processor array

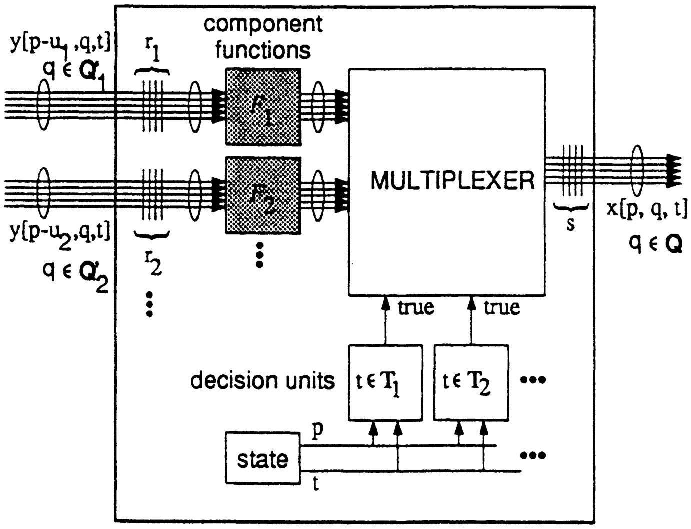


Fig. 12. Part of a processing element with modules

can be specified by an OCCAM program.

EXAMPLE 8. *The piecewise regular program for FIR-filtering in Example 7 is given. Let $p = j$ be the processor index and let $t = i$ be the sequencing index. Moreover, let us suppose that during the design a hierarchical structure has been generated. The following program may have been obtained:*

```
PROFIR
  in
    $(\langle \| p:\ 0 \le p < N :: a'[0,p]\rangle, \langle \| t:\ 0 \le t < T :: u'[t,0]\rangle)$
  always
    $\langle \| p:\ p = 0 :: \langle \| t:\ 0 \le t < T ::$
      $y[t,p] = a[t,p]u[t,p] \;\|$
      $a[t,p] = a[t-1,p] \quad \text{if} \quad t > 0$
      $\qquad\;\; \sim a'[t,p] \quad \text{if} \quad t = 0 \;\|$
      $u[t,p] = u'[t,p]$
    $\rangle\rangle$
    $\langle \| p:\ 0 < p < N ::$
      $(\langle \| t:\ 0 \le t < T :: y[t,p]\rangle, \langle \| t:\ 0 \le t < T :: u[t,p]\rangle) =$
        $PE(\langle \| t:\ 0 \le t < T :: y[t,p-1]\rangle,$
          $\langle \| t:\ t = 0 :: a'[t,p]\rangle, \langle \| t:\ 0 \le t < T :: u[t,p-1]\rangle)$
    $\rangle$
  out
    $\langle \| t:\ N-1 \le t < T :: y[t,N-1]\rangle$
```

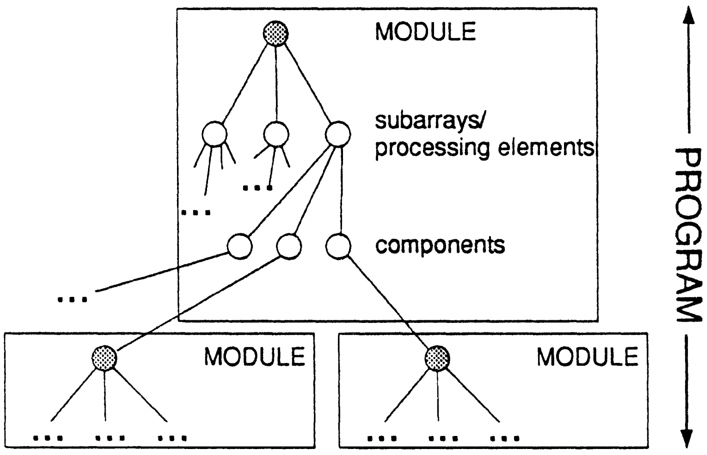


Fig. 13. Hierarchical composition of modules and its interpretation

```
PE
  in
    $(\langle \| t : 0 \le t < T :: y'[t] \rangle, \langle \| t : t = 0 :: a'[t] \rangle,$
      $\langle \| t : 0 \le t < T :: u'[t] \rangle)$
  always
    $\langle \| t : 0 \le t < T ::$
      $y[t] = y'[t] + a[t]u'[t] \;\|$
      $a[t] = a[t-1] \quad \text{if} \quad t > 0$
            $\sim a'[t] \quad \text{if} \quad t = 0 \;\|$
      $u[t] = u'[t]$
    $\rangle$
  out
    $(\langle \| t : 0 \le t < T :: y[t] \rangle, \langle \| t : 0 \le t < T :: u[t] \rangle)$
```

The processor consists of N processing elements of two kinds. The first processor space $\mathbf{P}_1$ is given by $p = 0$, the second processor space $\mathbf{P}_2$ is given by $0 < p < N$. The corresponding subarrays contain the processing elements $S_1[p]$ and $S_2[p]$. The interpretation as a signal flow graph is given in Fig.14. The example is used to demonstrate the above hardware interpretation scheme with no concern about efficiency. For example, the multiplexers may be omitted by using static registers to store the local variables. Moreover, pipelining could have been introduced using an affine transformation of the iteration indices.

5.3. Program Transformations and Design Flow

Until now, we have described an equational program notation and its interpretation as an algorithm (behavior) and as a processor architecture (structure). The

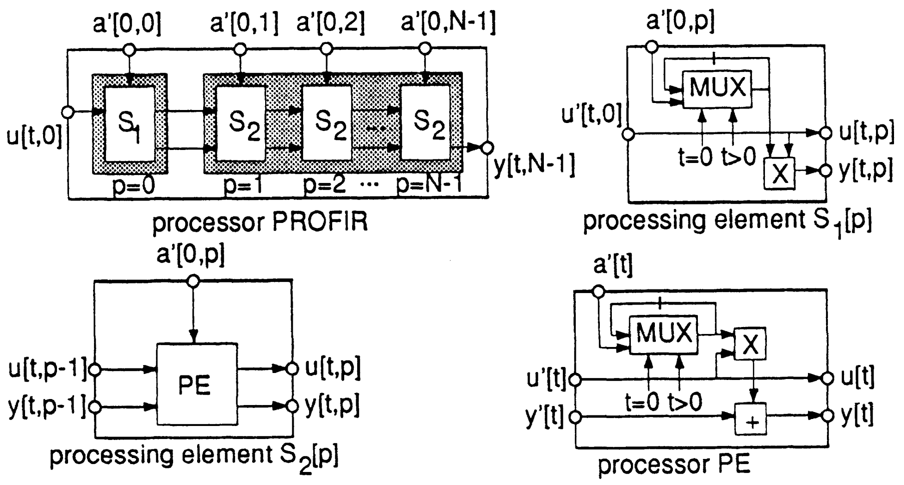


Fig. 14. Processor corresponding to the modules PROFIR and PE

program which specifies the structure must obey the given architectural specifications, e.g. the number of hierarchical levels, predefined (already realized) modules, size and dimension of processor spaces, timing constraints, synchronous vs. asynchronous realization, local control scheme, local data interconnection and many others.

One possibility to obtain a program which is computationally equivalent to the given one is to apply program transformations. These transformations must be able to solve the design problems mentioned in Section 4.1. The definitions and properties of linearly bounded lattices and piecewise linear/regular algorithms lead to a transformative approach with the following properties:

- The tools make use of basic program transformations which are provably correct, i.e. the input/output behavior is preserved.
- The class of piecewise linear programs is closed under the basic transformations. Therefore, the program can be processed further after any of the design transformations.

Because of space limitations it is not possible to describe all available tools and basic transformations. Moreover, many results have been published elsewhere. Therefore, the main purpose of this Section is to give some of the basic transformations and to more informally describe the tools.

5.3.1. Basic Transformations

Only the basic transformations are applied directly to a program. All other tools can perform program transformations only by calling these basic routines. As not all basic transformations can be explained in detail, we restrict ourselves to 'output normal form', 'unidirectional propagation', 'variable splitting' and 'piecewise affine transformation'. The implementation of these transformations necessitates access to a library of mathematical routines, which includes operations on lattices (change of basis, intersection, union), operations on polyhedra (linear programming, convex hull, integer linear programming, projection), integer linear algebra (change of basis, unimodular transformations, Hermite and Smith Normal form) and exact rational arithmetic, see e.g. Fig.5.

Output Normal Form: Here, the index function of a variable is transformed onto a normalized form. Therefore, the geometrical representation of the dependence graph is not changed. In particular, let us suppose a quantification of the form

$$quant = \langle \| \; I : \; I \in \mathbf{I} :: \; x[PI + f] = \mathcal{F}(..., y[QI - d], ...)\rangle$$

The ouput normalized form is obtained as

$$\begin{aligned} &OutputNormalForm(quant, x) = \\ &\quad \langle \| \; I : \; I \in \mathbf{I}' :: \; x[I] = \mathcal{F}(..., y[Q'I - d'], ...)\rangle \end{aligned}$$

where $\mathbf{I}'$ is an affine transformed index space, i.e. $\mathbf{I}' = f(\mathbf{I})$ with $f(I) = PI + f$. Moreover, Q' satisfies $Q'P = Q$ and $d' = Q'f + d$.

This transformation can be used to achieve a normalized input for other tools like localization, control generation or partitioning. There are many other transformations which just restructure the given program without actually changing the dependence graph, e.g. splitting of quantifications or removing and creating condition spaces.

EXAMPLE 9. *As an example we consider*

$$quant = \langle \| \; i : \; 1 \leq i \leq n :: \; x[2i - 3] = y[5i + 6]\rangle$$

With $P = 2$, $f = -3$, $Q = 5$, $d = 6$ *we obtain*

$$\begin{aligned} &OutputNormalForm(quant, x) = \\ &\quad \langle \| \; i : \; i = 2\kappa - 3 \; \wedge \; 1 \leq \kappa \leq n :: \; x[i] = y[\tfrac{5}{2}i + \tfrac{27}{2}]\rangle \end{aligned}$$

Unidirectional Propagation: The conversion of piecewise linear into piecewise regular programs is based on this program transformation. The main purpose is to replace linear index functions by constant index functions. In particular, let us suppose that the quantified equation

$$quant = \langle \| \; I : \; I \in \mathbf{I} :: \; x[I] = y[QI - d]\rangle$$

is given where $I \in \mathbf{Z}^s$ and $Q \in \mathbf{Q}^{s \times s}$. Moreover it is required that $rank(Q) = s - 1$. As a result, we obtain

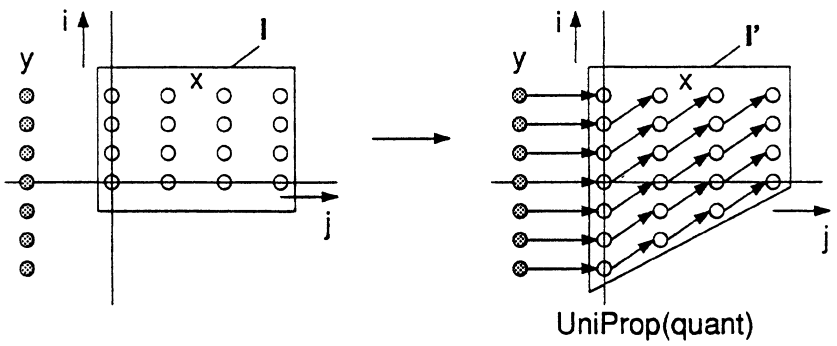


Fig. 15. Dependence graph representing an unidirectional propagation

$$UniProp(quant) = \langle \| I : I \in \mathrm{I}' :: x[I] = y[I-d] \text{ if } I \in \mathrm{I}^1 \sim x[I-u] \text{ if } I \in \mathrm{I}^2 \sim x[I+u] \text{ if } I \in \mathrm{I}^3 \rangle$$

In order to determine the parameters, we decompose the matrix $E - Q$, where E denotes the unity matrix, according to $E - Q = u\beta^t$ where $u \in \mathbf{Z}^s$ with coprime elements and $\beta \in \mathbf{Q}^s$. Then the spaces $\mathrm{I}^1 = \{I : \beta^t I = 0\}$, $\mathrm{I}^2 = \{I : \beta^t I > 0\}$, $\mathrm{I}^3 = \{I : \beta^t I < 0\}$ and $\mathrm{I}' = ConvexHull(\mathrm{I} \cup \{J : J = QI \wedge I \in \mathrm{I}\})$ are obtained.

EXAMPLE 10. *As an example*

$$quant = \langle \| i,j : i = \kappa_1 \wedge j = 2\kappa_2 \wedge 0 \le \kappa_1, \kappa_2 < 4 :: x[i,j] = y[i - \tfrac{j}{2}, -3] \rangle$$

is considered. Using $Q = \begin{pmatrix} 1 & -1/2 \\ 0 & 0 \end{pmatrix}$, $d = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $u^t = (\,1 \;\; 2\,)$ *and* $\beta^t = (\,0 \;\; 1/2\,)$ *we obtain*

$$\begin{aligned} UniProp(quant) = \\ \langle \| i,j : i = \kappa_1 \wedge j = 2\kappa_2 \wedge 0 \le \kappa_2 < 4 \wedge \kappa_2 - 4 < \kappa_1 < 4 :: \\ x[i,j] &= y[i, j-3] \quad \text{if} \quad j = 0 \\ &\sim x[i-1, j+2] \quad \text{if} \quad j > 0 \\ &\sim x[i+1, j-2] \quad \text{if} \quad j < 0 \rangle \end{aligned}$$

The corresponding dependence graphs are shown in Fig.15.

Variable Splitting: Let us suppose that a given program defines an indexed variable, say x, on a certain index space, say I. It is often required to rename this variable in different portions of the index space, say x_1 in J and x_2 in K. There are two main applications of variable splitting:

- In certain algorithms, e.g. algorithms for matrix inversion, SVD, algebraic path problem, the data dependencies are not regular. In order to determine feasible scheduling functions it is necessary to schedule different parts of the index space differently. Therefore, variables must be renamed.
- If additional hierarchical levels are created, then it is useful to make variables local to the new 'sub'-modules. To this end, these variables must not be used in any other part of the program. This property can be achieved by variable splitting.

In particular, let us assume that

$$\begin{aligned} quant = &\langle \| \, I : \; I \in \mathbf{I} :: \; x[I] = S[I] \rangle \; \| \\ &\langle \| \, I : \; I \in \mathbf{J} :: \; y[I] = \mathcal{F}(..., x[f(I)], ...)\rangle \end{aligned}$$

where x is defined by the first quantification in 'quant' only and the second quantification in 'quant' represents all quantifications in the module which use the variable x. $f(I)$ denotes an index function. Then

$$\begin{aligned} &VariableSplit(quant, \mathbf{I}_{cut}, x) = \\ &\quad\langle \| \, I : \; I \in \mathbf{I} \cap \mathbf{I}_{cut} :: \; x_1[I] = S[I] \rangle \; \| \\ &\quad\langle \| \, I : \; I \in \mathbf{I} \cap \overline{\mathbf{I}_{cut}} :: \; x_2[I] = S[I] \rangle \; \| \\ &\quad\langle \| \, I : \; I \in \mathbf{J} :: \; y[I] \; = \; \mathcal{F}(..., x_1[f(I)], ...) \;\; \text{if} \;\; I \in \mathbf{I}_1 \\ &\qquad\qquad\qquad\qquad\quad \sim \; \mathcal{F}(..., x_2[f(I)], ...) \;\; \text{if} \;\; I \in \mathbf{I}_2 \rangle \end{aligned}$$

Here, $\mathbf{I}_{cut}$ denotes the halfplane defining the index space of x_1, i.e. $\mathbf{I}_{cut} = \{I : aI \geq b\}$ and $\overline{\mathbf{I}_{cut}} = \{I : aI < b\}$ for some vector a and some scalar b. In order to guarantee input-ouput equivalence, we have $\mathbf{I}_1 = \{I : af(I) \geq b\}$ and $\mathbf{I}_2 = \{I : af(I) < b\}$.

EXAMPLE 11. *Let us assume that*

$$\begin{aligned} quant = &\langle \| \, i,j : \; 0 \leq i,j < N :: \; x[i,j] = z[i,j] \rangle \; \| \\ &\langle \| \, i,j : \; 0 \leq i,j < N :: \; y[i,j] = x[i+j, i-j] \rangle \end{aligned}$$

and $\mathbf{I}_{cut} = \{i,j : \; i - j \geq 0\}$. *Then we obtain (see Fig.16)*

$$\begin{aligned} &VariableSplit(quant, \mathbf{I}_{cut}, x) = \\ &\quad\langle \| \, i,j : \; 0 \leq i < N \; \wedge \; i \leq j < N :: \; x_1[i,j] = z[i,j] \rangle \; \| \\ &\quad\langle \| \, i,j : \; 0 \leq i < N \; \wedge \; 0 \leq j < i :: \; x_2[i,j] = z[i,j] \rangle \; \| \\ &\quad\langle \| \, i,j : \; 0 \leq i,j < N :: \; y[i,j] \; = \; x_1[i+j, i-j] \;\; \text{if} \;\; j \geq 0 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \sim \; x_2[i+j, i-j] \;\; \text{if} \;\; j < 0 \rangle \end{aligned}$$

Piecewise Affine Transformation: The affine transformation of index spaces is one of the most important basic transformations. The geometrical representation of the program's dependence graph is changed. This transformation has been used in many different context, e.g.

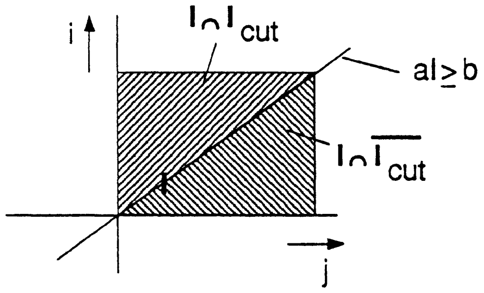


Fig. 16. Dependence graph representing variable splitting

- Parallelization of loop programs (loop skewing, do-accross, cycle shrinking, linear schedule).
- Synthesis of systolic arrays by affine transformations.
- Retiming transformations in synchronous and asynchronous circuits and processor arrays.

In general, the affine transformation of index spaces can be applied to each variable in a module separately without leaving the class of piecewise linear programs. Let us use the simple quantifications

$$\begin{aligned} quant = & \langle \| \, I : \; I \in \mathbf{I} :: \; x[f(I)] = S[I] \rangle \, \| \\ & \langle \| \, I : \; I \in \mathbf{J} :: \; y[I] = \mathcal{F}(..., x[g(I)], ...) \rangle \end{aligned}$$

where x is defined by the first quantification in 'quant' only and the second quantification in 'quant' represents all quantifications in the module which use the variable x. $S[I]$ denotes an expression and $f(I)$, $g(I)$ denote index functions. Then with the affine transformation $h(I) = \Lambda I + \gamma$ we obtain

$$\begin{aligned} AffineMap(quant, h, x) = & \langle \| \, I : \; I \in \mathbf{I} :: \; \hat{x}[h(f(I))] = S[I] \rangle \, \| \\ & \langle \| \, I : \; I \in \mathbf{J} :: \; y[I] = \mathcal{F}(..., \hat{x}[h(g(I))], ...) \rangle \end{aligned}$$

Obviously, the variable $x[J]$ has been replaced by $\hat{x}[h(J)]$. Therefore, the location of a certain value $x[J]$ in the geometrical representation of the dependence graph has been moved from J to $h(J)$. In order to guarantee I/O equivalence, $h(J)$ must be one-to-one for all index points J where x is defined.

If the transformation matrix Λ is identical for all variables of a module, the transformation used by Rao [9] for designing systolic arrays is obtained. Then the class of regular iterative algorithms is mapped onto itself. Condition which guarantee this property for general affine transformations are given in [6].

After an affine transformation, often a transformation to the output normal form is applied. The new indices may then be interpreted as processor index p and sequencing index t, i.e. $I^t = (\; p^t \;\; t \;)$. Certain restrictions on the affine transformations guarantee the causality of the resulting processor array.

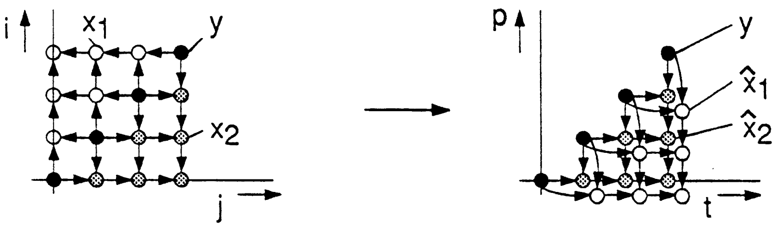


Fig. 17. Dependence graph representing piecewise affine transformations

EXAMPLE 12. *Let us start with the quantified equations*

$$
\begin{aligned}
quant = \langle \| \; i,j : \; 0 \le i,j < N :: & \\
x_1[i,j] &= y[i-1,j] \circ y[i,j+1] \quad \text{if} \quad i = j+1 \\
&\sim x_1[i-1,j] \circ x_1[i,j+1] \quad \text{if} \quad i > j+1 \\
x_2[i,j] &= y[i+1,j] \circ y[i,j-1] \quad \text{if} \quad i = j-1 \\
&\sim x_2[i+1,j] \circ x_2[i,j-1] \quad \text{if} \quad i < j+1 \rangle
\end{aligned}
$$

This partitioning of variables may have been obtained by variable splitting, see Fig.17. It can be seen, that a linear schedule of the program given above can not be obtained. We choose a piecewise affine transformation using $h_1(I) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} I$ *for transforming* x_1 *and* $h_2(I) = I$ *for* x_2*. After transforming the resulting quantification onto output normal form and renaming the variables according to* $I^t = (\; p \;\; t \;)$ *a quantification is obtained which can be interpreted as a processor as follows*

$$
\begin{aligned}
\langle \| \; p,t : \; 0 \le p < N \;\wedge\; p \le t < N :: & \\
\hat{x}_1[p,t] &= y[p,t-1] \circ y[p+1,t] \quad \text{if} \quad p = t-1 \\
&\sim \hat{x}_1[p,t-1] \circ \hat{x}_1[p+1,t] \quad \text{if} \quad p < t-1 \\
\hat{x}_2[p,t] &= y[p+1,t] \circ y[p,t-1] \quad \text{if} \quad p = t-1 \\
&\sim \hat{x}_2[p+1,t] \circ \hat{x}_2[p,t-1] \quad \text{if} \quad p < t-1 \rangle
\end{aligned}
$$

5.3.2. Tools

This last section is devoted to some of the tools shown in Fig.5. In particular, the basic problems of localization, control generation, processing of hierarchy and partitioning are introduced and the corresponding procedures are briefly described.

Localization and Control Generation: The localization of algorithms and the generation of control signals for the synchronization of parallel computations are part of the whole design trajectory. We will summarize the main results only. For further information and a complete list of references, the interested reader is referred to [17, 33, 34].

The *localization* of algorithms converts piecewise linear programs into piecewise regular programs. Main applications of the localization are the distribution of control signals and the distribution of data to the locations requiring them. Moreover, the class of piecewise linear programs is considerably more general than that of piecewise regular programs.

The localization can also be considered as a routing problem in the index space. The following problems can be solved explicitly, see also [31, 32]:

- Perform an asymptotically optimal localization with respect to the number and the total lengths of interconnections between processing elements.
- Consider a given set of data dependence vectors or interconnection directions.
- Solve localization problems which are due to an unbounded number of equations requiring a certain value (unbounded out-degree) or due to an unbounded distance between the index point, where a variable is defined and the index point where the variable is used. Moreover, even global commutative and associative operations (like summation, multiplication, min, max) can be transformed into sequences of simple operations mechanically, see [33].
- The localization can be applied to the modules of a program independently.

The localization is based on the above given 'unidirectional propagation'. Obviously, this basic transformation is able to convert a quantification of the form

$$\langle \| \, I : \; I \in \mathrm{I} :: \; x[PI] = y[QI] \rangle$$

into a quantification with constant index functions, if $rank(P - Q) = 1$. The general case of an affine index function with $rank(P - Q) > 1$ can be obtained by a suitable decomposition of $P - Q$ into a summation of outer vector products.

The purpose of *control generation* is to replace completely or partly the iteration dependent conditionals. This is achieved as follows:

- Additional control variables are defined which determine the actual functions that must be evaluated within the processing elements. Iteration dependent conditionals are replaced by conditionals which involve the introduced control variables.
- The definition of these variables at all index points and the definition of initial values are included in the given program in form of a set of quantified equations.
- As these quantified equations have constant index functions, the control signals are propagated through the final processor array.
- The resulting program can be processed further as it is in the class of piecewise regular programs.

Following the structural interpretation of a program proposed in Section 5.2.4, iteration dependent conditionals are implemented by decision units whose inputs necessitate the internal state of a processing element. In general, these decision units must be capable of e.g. addition, multiplication, counting and comparison in order to decide whether $t \in \mathrm{T}_i$. The control generation replaces iteration dependent conditionals by conditionals which depend on control variables. Now, the

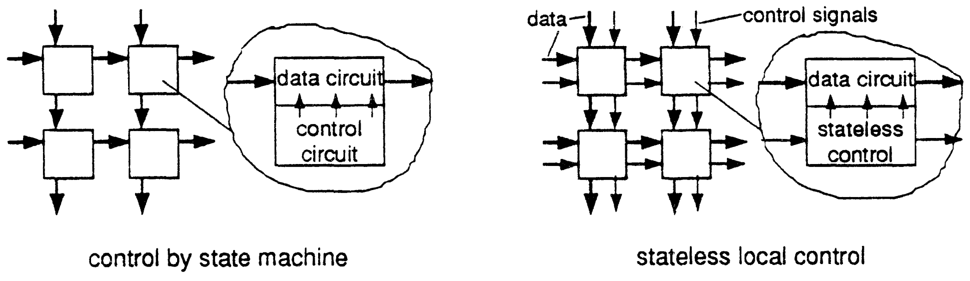


Fig. 18. Stateless local control mechanism

decision units evaluate combinatorial functions of the control variables, see Fig.18. Hence, the control generation leads to a completely stateless control mechanism and the processing elements are independent of the size of the problem to be solved. The main purpose of the *control generation* is to simplify the design process. For example, it is possible to apply the control generation to all condition spaces and to all modules. As a result, a completely regular program (e.g. one that represents a regular iterative algorithm) is obtained which can be processed further more easily.

Creation and Flattening of Hierarchy: There are several reasons to process hierarchy in the design of massive parallel architectures:

- Usually, the behavior of the required architecture is described by a hierarchical program. The partitioning of the program in hierarchical levels has been done in the algorithm design step of the design trajectory, see Fig.1. But for the processor specification, usually a different hierarchical partitioning is required as the architecture specification must be taken into account. Therefore, the creation of new hierarchical levels or the flattening of a given structure are important design steps.
- Hierarchy enables a decomposition of the whole design problem into simpler subproblems. Details of the design or the specification which are not necessary at the current stage can be hidden and tools can be applied to several hierarchical levels independently.
- One of the most important design transformations, i.e. partitioning or hardware matching, can be considered as a hierarchical operation.

It is of great importance that tools which create and flatten hierarchy fit into the design methodology, i.e. they must relate piecewise linear programs and they must be provably correct. In the following, only an informal introduction is given, for a more complete treatment, see [57].

At first, the *creation of hierarchical levels* is described. The purpose of this tool is to include a set of equations into a new module. After the creation of the new module, these equations are hidden from the original module point of view. In the following, the module which uses the function defined by the new module is the 'calling' or 'main' module whereas the newly created module is the 'called' module or 'sub'-module.

The procedure for creating additional hierarchical levels is based on the interpretation of index directions as hierarchical levels. It is described for a simple input program only. More complicated cases can be handled in a similar way.

1. At first the basic transformations 'variable splitting', 'affine transformation' and 'normal form computation' are applied. This step enables the inclusion of a desired set of equations in the new module.
2. Let us assume that the equations defined by the quantification
$$\langle \| \; I : \; I \in \mathbf{I} :: \; x[f(I)] = \mathcal{F}(..., y[h(I)], ...)\rangle$$
shall be included in the new module. To this end, the index space I is decomposed according to $I = \begin{pmatrix} J \\ K \end{pmatrix}$ into $\langle \| \; J : \; J \in \mathbf{J} :: \; \langle \| \; K : \; K \in \mathbf{K} :: \; ...\rangle\rangle$. Consequently, the above given quantification can be written as
$$\begin{aligned}&\langle \| \; J : \; J \in \mathbf{J} :: \; \langle \| \; K : \; K \in \mathbf{K} :: \\ &\qquad x[f_1(J), f_2(K)] = \mathcal{F}(..., y[h_1(J), h_2(K)], ...)\rangle\rangle\end{aligned}$$
It can be seen that the index functions $f(I)$ and $h(I)$ have also been decomposed into components which depend on J only and those which depend on K only. This restriction on the class of quantified equations which can be included in a new module simplifies the following discussions.
3. The above quantified equation of the given module can now be replaced by
$$\begin{aligned}&\langle \| \; J : \; J \in \mathbf{J} :: \; (\langle \| \; K : \; K \in \mathbf{K} :: \; x[f_1(J), f_2(K)]\rangle) = \\ &\qquad sub(..., \langle \| \; K : \; K \in \mathbf{K} :: \; y[h_1(J), h_2(K)]\rangle, ...)\rangle\end{aligned}$$
which yields the calling module. Moreover, the newly created called module which defines the function 'sub' has the form
```
sub
  in
     (..., ⟨‖ K : K ∈ K :: y'[h₂(K)]⟩, ...)
  always
     ...
  out
     ⟨‖ K : K ∈ K :: x'[f₂(K)]⟩
```
If $f_2(K)$ or $h_2(K)$ are not one-to-one for $K \in \mathbf{K}$, then it is possible to restrict the quantifications to the corresponding unique subsets. This is due to the fact that it is not necessary to transfer the same value more than once to the new module and vica versa.

Some aspects of the mechanism will be explained using the next example.

EXAMPLE 13. *The example explains the above described procedure using a program for multiplying two matrices. Note that all index spaces are of the form $1 \leq i, j, k \leq n$. They are not included in the program text.*

```
mat1
  in
    $(\langle\| \; i,j :: \; a[i,j]\rangle \; , \; \langle\| \; i,j :: \; b[i,j]\rangle)$
  always
    $\langle\| \; i,j,k :: \; c[i,j,k] = c[i,j,k-1] + a[i,k]b[k,j] \quad \text{if} \quad k > 1$
                          $\sim a[i,k]b[k,j] \quad \text{if} \quad k = 1\rangle$
  out
    $\langle\| \; i,j :: \; c[i,j,n]\rangle$
```

Now, a new level for the inner product calculation is introduced. Obviously, the variables $c[i,j,k]$ can be made local to the module 'inpro'. This can be achieved by replacing $c[i,j,n]$ by a new variable, e.g. $\bar{c}[i,j]$, using variable splitting.

```
mat2
  in
    $(\langle\| \; i,j :: \; a[i,j]\rangle \; , \; \langle\| \; i,j :: \; b[i,j]\rangle)$
  always
    $\langle\| \; i,j,k :: \; \bar{c}[i,j] \; = \; c[i,j,k] \quad \text{if} \quad k = n \; \|$
                  $c[i,j,k] \; = \; c[i,j,k-1] + a[i,k]b[k,j] \quad \text{if} \quad k > 1$
                           $\sim \; a[i,k]b[k,j] \quad \text{if} \quad k = 1\rangle$
  out
    $\langle\| \; i,j :: \; \bar{c}[i,j]\rangle$
```

Now, the above described decomposition of the index space according to $\langle\| \; i,j,k :: \; ...\rangle = \langle\| \; i,j :: \; \langle\| \; k :: \; ...\rangle\rangle$ leads to the final result

```
mat3
  in
    $(\langle\| \; i,j :: \; a[i,j]\rangle \; , \; \langle\| \; i,j :: \; b[i,j]\rangle)$
  always
    $\langle\| \; i,j :: \; \bar{c}[i,j] = inpro(\langle\| \; k :: \; a[i,k]\rangle, \langle\| \; k :: \; b[k,j]\rangle)\rangle$
  out
    $\langle\| \; i,j :: \; \bar{c}[i,j]\rangle$
```

```
sub
  in
    $(\langle\| \; k :: \; a'[k]\rangle \; , \; \langle\| \; k :: \; b'[k]\rangle)$
  always
    $\langle\| \; k :: \; c' = \hat{c}[k] \quad \text{if} \quad k = n$
          $\hat{c}[k] = \hat{c}[k-1] + a'[k]b'[k] \quad \text{if} \quad k > 1$
                $\sim a'[k]b'[k] \quad \text{if} \quad k = 1\rangle$
  out
    $c'$
```

Now, *flattening of hierarchical levels* is described. This transformation is the inverse of the above defined operation. Therefore, a module which defines a function that is used by other 'calling' modules can be removed from the program by including its equations into these calling modules. Again, the process of flattening hierarchical levels is explained by means of a simple case only.

1. The main concept of the proposed procedure is based on the fact that all variable names used within the called module must be different from those used in calling modules before the equations can be included. In particular, if the called module is used in several equations of several calling modules, copies of the called module with disjoint variable names must be generated.
2. The above described uniqueness of variables must also be guaranteed for quantified uses of the called module. This can be achieved as follows: if a module is used for all $J \in \mathbf{J} \subset \mathbf{Z}^s$ then to all variables in the called module, s additional index dimensions are added. This transformation must be applied to variables in the 'in' and 'out' sections of the called module also. The extensions of the index spaces can be interpreted as an embedding of the variables in the index spaces of the calling modules.
3. Finally, the equations of the called module are included in the calling modules. Additional quantified equations consider the former interface between calling and called modules.

Therefore, if module which defines the function 'sub' is used by a calling module according to

$$\begin{aligned}&\langle \| J : J \in \mathbf{J} :: (\langle \| K : K \in \mathbf{K}_1 :: x[f(J,K)]\rangle) = \\ &\qquad sub(\ldots, \langle \| K : K \in \mathbf{K}_2 :: y[g(J,K)]\rangle, \ldots)\rangle\end{aligned}$$

and the called module 'sub' has the form

```
sub
  in
    $\langle \| K : K \in \mathbf{K}_2 :: y'[g'(K)]\rangle$
  always
    ...
  out
    $\langle \| K : K \in \mathbf{K}_1 :: x'[f'(K)]\rangle$
```

then the following quantified equations must be included in the calling module if the called module 'sub' is removed from the program:

$$\begin{aligned}&\langle \| J : J \in \mathbf{J} :: \\ &\qquad \text{always section of 'sub' with embedded variables } \| \\ &\qquad \langle \| K : K \in \mathbf{K}_1 :: x[f(J,K)] = x'[J, f'(K)]\rangle \; \| \\ &\qquad \langle \| K : K \in \mathbf{K}_2 :: y'[J, g'(K)] = y[g(J,K)]\rangle\rangle\end{aligned}$$

Again, this procedure is explained by an example.

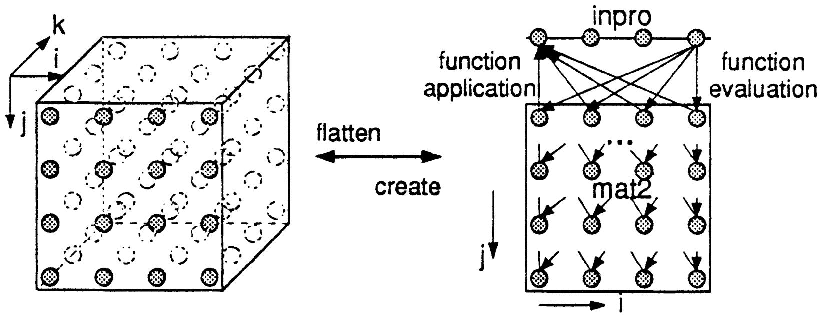


Fig. 19. Example of flatten and create hierarchy

EXAMPLE 14. *The hierarchy of the modules 'mat3' and 'inpro' defined in the previous example will be flattened. According to the procedure sketched above, variables in the called module 'inpro' must be made unique. As a', b' ,c' and $\bar{c}$ are not used in the module 'mat3', no new names must be introduced. Now, these variables are embedded in the index space of the calling module 'mat3' by replacing $a'[k]$, $b'[k]$, c' and $\bar{c}[k]$ with $a'[i,j,k]$, $b'[i,j,k]$, $c'[i,j]$ and $\bar{c}[i,j,k]$, respectively. The resulting equations can be included into the calling module 'mat3' which yields*

```
mat4
  in
      $(\langle \| \; i,j :: a[i,j] \rangle \; , \; \langle \| \; i,j :: b[i,j] \rangle)$
  always
      $\langle \| \; i,j :: \bar{c}[i,j] = c'[i,j] \; \|$
      $\langle \| \; k :: a'[i,j,k] \; = \; a[i,k] \; \| \; b'[i,j,k] = b[k,j] \; \|$
                  $c'[i,j] \; = \; \bar{c}[k,i,j] \quad \text{if} \quad k = n \; \|$
                  $\bar{c}[i,j,k] \; = \; \bar{c}[i,j,k-1] + a'[i,j,k]b'[i,j,k] \quad \text{if} \quad k > 1$
                              $\sim \; a'[i,j,k]b'[i,j,k] \quad \text{if} \quad k = 1 \rangle \rangle$
  out
      $\langle \| \; i,j :: \bar{c}[i,j] \rangle$
```

After a simple substitution of variables, the original program 'mat1' is obtained again. Fig.19 serves to explain the process of flattening and creating hierarchy.

Partitioning: Partitioning or hardware matching is a complex program transformation which enables the design of fixed size architectures. Usually, the architecture specification not only prescribes the number of available processing elements but also the dimension of the interconnection structure, see Fig.20. An overview of existing techniques is given in [58]. The procedures proposed up to date are based on the partitioning of the algorithm's iteration space into congruent tiles. Subsequently, these tiles are mapped on a processor array of fixed size.

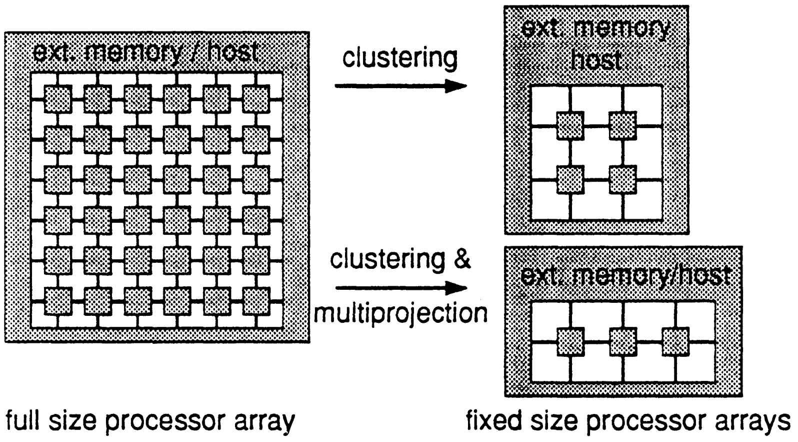


Fig. 20. Hardware Matching

This approach is due to the fact that the partitioning of a given processor array obviously is equivalent to a partitioning the program's index space. Jainandunsing [24] classifies different partitioning methods into LSGP (locally sequential, globally parallel) and LPGS (locally parallel, globally sequential) schemes. In case of the LSGP-clustering scheme, operations inside a tile are sequentialized and operations of distinct tiles are executed in parallel on an array of fixed size. In case of the LPGS-clustering scheme, operations of different tiles are executed sequentially on an array of fixed size, whereas operations inside a tile are executed by different processing elements in parallel. In [58], this classification is generalized by the introduction of the partitioning schemes of active clustering and passive clustering: whereas active clustering is concerned with the problem of a limited number of processing elements, passive clustering serves to increase the efficiency of a processor array. Moreover, multiprojection is a partitioning scheme for the reduction of the dimension of a processor array. In [21], a systematic way of deriving linear processor arrays is given. The problem of partitioning has also found theoretical interest in the domain of 'compilers for supercomputers'. In e.g. [23, 20, 28, 53], similar partitioning and multiprojection schemes are proposed for loop-programs.

The proposed piecewise linear design methodology enables a homogeneous and complete solution to the partitioning problem, see [29]. The approach can be summarized as follows:

- Instead of providing a large set of possible transformations for solving different problems, e.g., for realizing different partitioning schemes, only one new program transformation is introduced.
- This program transformation partitions the index space of the given program into the direct sum of lattices. *The created tiles are embedded in new dimensions of the index space of a so created intermediate program.*

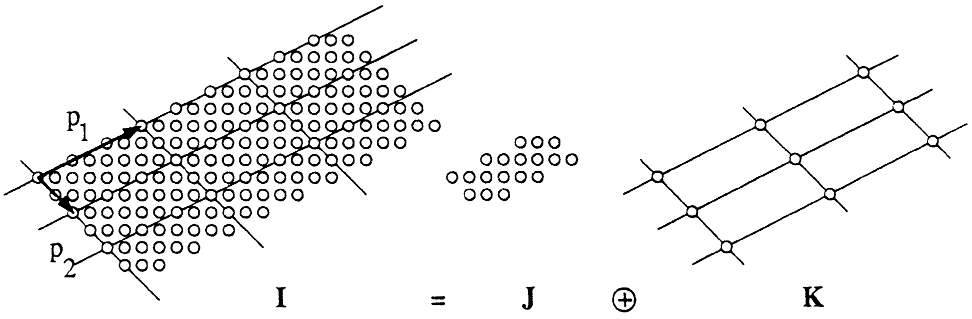


Fig. 21. Example of tiling an iteration space I according to $\mathbf{I} = \mathbf{J} \oplus \mathbf{K}$ for an iteration space of dimension 2

- After tiling, either an additional level of hierarchy can be introduced (which may be interpreted as a processing element) or an affine transformation is applied directly to the intermediate program. As a result, the dimension of the iteration space is reduced by scheduling the tiles and the operations inside a tile. *Different choices of coordinate directions which are associated to the processor space and the sequencing space enable the generalization and consideration of known partitioning schemes like multiprojection, LSGP-, LPGS-, active- and passive clustering.*
- The control of processor functions, the control of external memory access, the switching functions for data and control paths and the ordering of input and output data to and from the host are specified by the resulting partitioned program.

Let us shortly describe the main steps involved in the partitioning of piecewise regular programs, see [29]:

- First, the s-dimensional iteration space of the given piecewise regular program is partitioned into congruent tiles **J** (see Fig.21). To this end, a tiling matrix P is chosen which determines the cuts through the index space **I**.
- The dimension of the iteration space is increased such that at most s dimensions are associated to each tile **J** and at most s dimensions are associated to the repetition of tiles described by **K**. In terms of the index vector $I \in \mathbf{Z}^s$, a new index vector $\hat{I} = \begin{pmatrix} J \\ K \end{pmatrix}$ is generated. Hence, all variables are embedded in an at most $2s$-dimensional iteration space. As the relations between the variables of the given program must be preserved, additional quantifications are added that define 'terminals' between the operations on the subspaces **J** and **K**.

An example for the partitioning of a simple two-dimensional index space (e.g. that of the modules 'mavec' or 'PRFIR' defined in the examples of Section 5.2) is given in Fig.22.

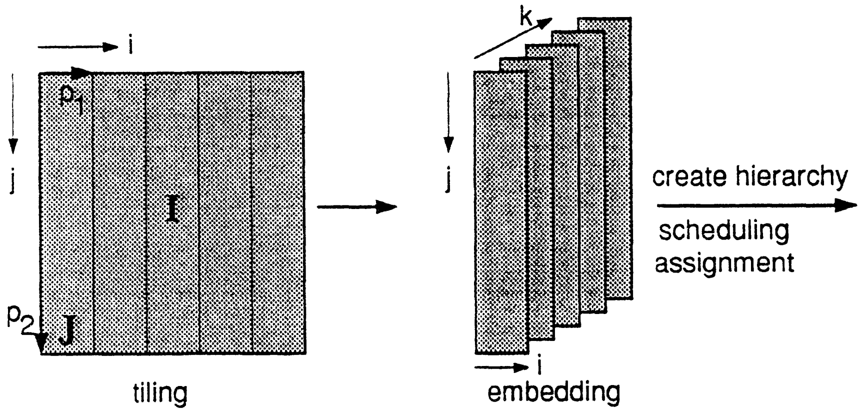


Fig. 22. Example of tiling and embedding an iteration space

The next step in the partitioning may be either the creation of a new hierarchical level or the scheduling and assignment of operations using a multiprojection, i.e. an affine transformation. In any case, the choice of coordinate directions generates all mentioned clustering and partitioning schemes *homogeneously*. For example, if in Fig.22 the directions i and j are assigned to a new module (or if they are scheduled sequentially and k is assigned to the processor index), then the LSGP-clustering scheme is obtained.

It has been shown, that the above given procedure to the hardware matching problem leads to a piecewise linear or piecewise regular program which can be processed further, see e.g. [29].

As a result of the whole section we can conclude that program transformations like partitioning, localization, control generation and piecewise affine scheduling fit into the piecewise linear design methodology. The class of piecewise linear programs is closed under these complex operations.

Acknowledgements

This research was partially supported by the 'Deutsche Forschungsgemeinschaft', project B5/SFB124 and by the Siemens AG. Moreover, the author is grateful to many interesting discussions on the subject of this paper with Ed Deprettere. The author would like to recognize the members of his group for suggestions and comments, especially J. Teich and U. Arzt.

References

[1] F. Catthoor, "Microcoded processor architectures and synthesis methodologies for real-time signal processing - a tutorial," in *Algorithms and Parallel VLSI-Architectures, Volume A* (E. F. Deprettere and A.-J. van der Veen, eds.), pp. 403–430, Amsterdam: Elsevier, 1991.

[2] H. Kung and C. Leiserson, "Systolic arrays for VLSI," in *SIAM Sparse Matrix Proceedings*, (Philadelphia), pp. 245–282, 1978.

[3] S. Y. Kung, *VLSI Processor Arrays.* Englewood Cliffs.: Prentice Hall, 1987.

[4] C. Guerra and R. Melhem, "Synthesizing non-uniform systolic designs," in *IEEE Int. Conf. Parallel Processing*, pp. 765–771, 1986.

[5] G. Mei, W. Liu, R. Cavin, and F. Lin, "Synthesizing irregular iterative algorithms with array architectures," *VLSI Signal Processing (eds. R.W. Brodersen et al.)*, vol. 3, pp. 447–458, 1988.

[6] L. Thiele, "On the design of piecewise regular processor arrays," in *Proc. IEEE Symp. on Circuits and Systems*, (Portland), pp. 2239–2242, 1989.

[7] L. Thiele, "On the hierarchical design of VLSI processor arrays," in *IEEE Symp. on Circuits and Systems*, (Helsinki), pp. 2517–2520, 1988.

[8] D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proceedings of the IEEE*, pp. 113–120, 1983.

[9] S. K. Rao, *Regular iterative algorithms and their implementations on processor arrays.* PhD thesis, Stanford University, 1985.

[10] H. Leverge, C. Mauras, and P. Quinton, "A language oriented approach to the design of systolic chips," in *Algorithms and Parallel VLSI-Architectures, Volume A* (E. F. Deprettere and A.-J. van der Veen, eds.), pp. 309–328, Amsterdam: Elsevier, 1991.

[11] U. Arzt and L. Thiele, "Hardware description with VLSI-Occam," in *Proc. IFIP 10th International Computer Hardware Description Languages*, (Marseille), April 1991.

[12] C. Lengauer, M. Barnett, and D. Hudson, "Towards systolizing compilation," *Distributed Computing*, vol. 5, pp. 7–24, 1991.

[13] K. Chandy and J. Misra, *Parallel Program Design.* Reading, Mass.: Addison-Wesley Publ. Comp., 1988.

[14] J. Snepscheut and J. Swenker, "On the design of some systolic algorithms," *Journal of the ACM*, pp. 826–840, 1989.

[15] J. A. Yang and Y. Choo, "Parallel program transformations using a metalanguage," in *Proc. ACM Conf. on Priciples of Programming Languages*, pp. 11–20, 1991.

[16] L. Thiele, "On the optimization of regular wavefront arrays," in *IEEE Conf. on Acoust., Speech, and Signal Processing*, (New York), pp. 2029–2032, 1988.

[17] J. Teich and L. Thiele, "Control generation in the design of processor arrays," *Int. Journal on VLSI and Signal Processing*, vol. 3, no. 2, pp. 77–92, 1991.

[18] A. D. Lange, A. V. D. Hoeven, P. Dewilde, and E. Deprettere, "HIFI: An object oriented system for the high level specification, analysis and synthesis of VLSI networks," in *Formal VLSI Specification and Synthesis* (L. Claesen, ed.), pp. 321–340, Amsterdam: North Holland, 1990.

[19] Y. Wong and J. Delosme, "Optimal systolic implementation of n-dimensional recurrences," in *Proc. IEEE Int. Conf. Computer Design*, pp. 618–621, 1985.

[20] P. Lee and Z. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. on Computers*, vol. 37, December 1988.

[21] U. Schwiegelshohn and L. Thiele, "Linear processor arrays for matrix computations," *J. on Parallel and Distributed Computing*, vol. 7, pp. 28–39, 1989.

[22] J. Xue and C. Lengauer, "On one-dimensional systolic arrays," in *Proc. ACM Int. Workshop n Formal Methods in VLSI Design*, (Springer Verlag), 1991.

[23] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. SIGPLAN*, (San Diego), pp. 319–329, Jan. 1988.

[24] K. Jainandunsing, "Optimal partitioning scheme for wavefront/systolic array processors," in *Proc. IEEE Symp on Circuits and Systems*, 1986.

[25] D. I. Moldovan and R. A. B. Fortes, "Partitioning and mapping of algorithms into fixed size systolic arrays," *IEEE Trans. Computers*, vol. C-35, pp. 1–12, 1986.

[26] J. Bu and E. Deprettere, "Processor clustering for the design of optimal fixed-size systolic arrays," in *Algorithms and Parallel VLSI-Architectures, Volume A* (E. F. Deprettere and A.-J. van der Veen, eds.), pp. 341–362, Amsterdam: Elsevier, 1991.

[27] J. Ramanujam and P. Sadayappan, "Tiling of iteration spaces for multicomputers," in *Int. Conf. on Parallel Processing*, pp. II/179–II/186, 1990.

[28] J.-P. Sheu and T.-H. Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 430–439, 1991.
[29] J. Teich and L. Thiele, "Partitioning of processor arrays: A piecewise regular approach," *INTEGRATION: The VLSI Journal*, 1992.
[30] S. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, pp. 88–105, 1989.
[31] Y. Wong and J. M. Delosme, "Broadcast removal in systolic algorithms," in *Proc. of Int'l Conf. on Systolic Arrays*, (San Diego), pp. 403–412, May 1988.
[32] V. V. Dongen and P. Quinton, "Uniformization of linear recurrence equations: A step toward the automatic synthesis of systolic arrays," in *Proc. Int'l Conf. Systolic Arrays*, (San Diego), pp. 403–412, 1988.
[33] V. Roychowdhury, L. Thiele, S. K. Rao, and T. Kailath, "On the localization of algorithms for VLSI processor arrays," *in: VLSI Signal Processing III, IEEE Press, New York*, pp. 459–470, 1989.
[34] L. Thiele and V. Roychowdhury, "Systematic design of local processor arrays for numerical algorithms," *in: Parallel Algorithms and VLSI Architectures: Volume A (E. Deprettere Ed.), North Holland Publishers*, pp. 329–339, 1990.
[35] H. Ribas, "Obtaining dependence vectors for nested loop programs," in *Proc. Int. Conference on Parallel Processing*, pp. II/212-II/219, 1990.
[36] J. Bu, L. Thiele, and E. Deprettere, "Systolic array implementation of nested loop programs," in *Application Specific Array Processors, Princeton*, (IEEE Computer Society Press), pp. 31–43, Sept. 1990.
[37] P. Frison, P. Gachet, and P. Quinton, "Designing systolic arrays with diastol," in *VLSI SIGNAL PROCESSING II*, pp. 93–105, IEEE Press, New York, 1986.
[38] P. Gachet, B. Joinnault, and P. Quinton, "Synthesizing systolic arrays using DIASTOL," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquart, eds.), pp. 25–36, Hilger, 1987.
[39] J. Jean and S. Kung, "A VLSI array compiler (VACS) for array design," in *Proc. Int. Conf. on VLSI and Signal Processing, VLSI Signal Processing III, IEEE Computer Society Press*, (Monterey), p. 1988, 495–508.
[40] B. Engstrom and P. Capello, "The SDEF systolic programming system," in *Concurrent Computations* (S. Tewksbury, B. Dickinson, and S. Schwarz, eds.), p. chapter 15, Plenum Press, 1987.
[41] D. Moldovan, "ADVIS: a software package for the design of systolic arrays," in *IEEE Conf. Computer Design*, pp. 158–164, 1984.
[42] V. V. Dongen and M. Petit, "PRESAGE: a tool for the parallelization of nested lopp programs," in *Formal VLSI Specification and Synthesis, volume 1* (L. Claesen, ed.), pp. 341–360, Amsterdam: North Holland, 1990.
[43] U. Arzt, J. Teich, and L. Thiele, "The concepts of COMPAR: A compiler for massive parallel architectures," in *Proc. International Symposium on Circuits and Systems (ISCAS)*, (San Diego), May 1992.
[44] J. Annevelink and P. Dewilde, "HIFI: A functional design system for VLSI processing arrays," in *Proc. Int'l Conf. on Systolic Arrays*, (San Diego), pp. 413–452, 1988.
[45] M. V. Swaaij, J. Rossel, F. Catthoor, and H. D. Man, "Synthesis of ASIC regular arrays for real-time image processing systems," in *Algorithms and Parallel VLSI-Architectures, Volume B* (E. F. Deprettere and A.-J. van der Veen, eds.), pp. 329–341, Amsterdam: Elsevier, 1991.
[46] J. Bu and E. F. Deprettere, "A design methodology for fixed-size systolic arrays," *Proceedings Conference on Application Specific Array Processors*, vol. ASAP90, pp. 591,602, 1990.
[47] R. Kuhn, "Transforming algorithms for single-stage and VLSI architectures," *Workshop Interconnection Networks for Parallel and Distributed Processing*, 1980.
[48] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *The IEEE/ACM 11-th Annual Int'l Symp. on Computer Architecture*, (Ann Arbor, MI, USA), pp. 208–214, 1984.
[49] W. L. Miranker and A. Winkler, "Space-time representation of computational structures," *Computing*, pp. 93–114, 1984.
[50] P. Capello and K. Steiglitz, "Unifying VLSI array design with linear transformations of

space-time," in *Advances in Computing Research*, vol. 2, pp. 23–65, 1984.
[51] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementations on processor arrays," *Proceedings of the IEEE*, vol. 6, pp. 259–282, March 1988.
[52] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, 1967.
[53] M. Wolfe and M.S.Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 452–471, 1991.
[54] M. Wolfe, "Massive parallelism through program restructuring," in *Proc. IEEE Conf. on Frontiers of Massively Parallel Computation*, pp. 407–415, 1990.
[55] L. Thiele, "Computational arrays for Jacobi algorithms," in *SVD and Signal Processing*, pp. 369–383, North Holland Pub., 1988.
[56] J. Teich and L. Thiele, "Uniform design of parallel programs for DSP," in *Proc. IEEE Int. Symp. Circuits and Systems*, (Singapore), pp. 344a–347a, June 1991.
[57] U. Arzt, J. Teich, and L. Thiele, "Hierarchical concepts in the design of processor arrays," in *Proc. CompEuro 1992*, (The Hague), May 1992.
[58] J. Bu, *Systematic design of regular VLSI processor arrays*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 1990.
[59] G. Nemhauser and L. Wolsey, *Integer and combinatorial optimization*. New York: John Wiley and Sons, 1988.

Appendix

This appendix contains methods to determine the solution of integer linear equations, see e.g. [59]. At first, we need some definitions and notations:

DEFINITION 2. *An integer matrix* $U \in Z^{s\times s}$ *is unimodular iff* $|\det U| = 1$.

It directly follows, that the inverse of a unimodular matrix is an integer matrix. In the next definition, the expression $a|b$ denotes a divides b.

DEFINITION 3. *Any matrix* $A \in Z^{s\times l}$ *has a decomposition of the form*

$$A = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V$$

where $U \in Z^{s\times s}$ *and* $V \in Z^{l\times l}$ *are unimodular. Moreover,* $D = diag(\delta_1, ..., \delta_r) \in Z^{r\times r}$ *with* $\delta_1|\delta_2|...|\delta_r$. $\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}$ *is called* Smith Normal Form *of* A.

Now, the set of integer solutions to a set of linear equations, i.e.

$$x \in \{x : Ax = b \wedge x \in Z^l\}$$

can be determined, see e.g. [59]. To this end, we compute the Smith Normal Form of A as

$$A = (\, U_1 \;\; U_2 \,) \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}$$

Using the decompositions $U^{-1} = \begin{pmatrix} U_1^- \\ U_2^- \end{pmatrix}$ and $V^{-1} = \begin{pmatrix} V_1^- & V_2^- \end{pmatrix}$ we obtain the following result:

- If $U_2^- b \neq 0$ then $Ax = b$ has no solution.
- If $D^{-1}U_1^- b$ is not integral, then $Ax = b$ has no integral solution.
- Otherwise, all integer solutions of $Ax = b$ are obtained using

$$x \in \{x : \; x = \begin{pmatrix} V_1^- & V_2^- \end{pmatrix} \begin{pmatrix} D^{-1}U_1^- b \\ \kappa \end{pmatrix} \; \wedge \; \kappa \in Z^{l-r}\}$$

For computational purposes it may be advantageous to use the Hermite Normal Form instead of the Smith Nomal Form.

PROGRAMMABLE CELLULAR NEURAL NETWORKS A STATE–OF–THE–ART

TAMÁS ROSKA
Dual and Neural Computing Systems Research Laboratory
Computer and Automation Institute, Hungarian Academy of Sciences (MTA SzTAKI)
Kende-u. 13/17, Budapest, H-1111, Hungary

Abstract. Analog processor arrays placed on a 3D regular grid interacting within a finite neighborhood: this is the CNN (cellular neural network) paradigm. Using other words: it is a programmable prototype machine performing nonlinear dynamic spatial convolutions in real time. The first silicon chips show enormous computing power: in the order of tera XPS (analog operations per second). In this paper the state of the art of this rapidly emerging field of computing is exposed emphasizing the programmability aspects. After describing the general framework, qualitative results are summarized. The application capabilities are shown in three selected areas: motion detection, retina models, and layout error detection. The hardware realizations are reviewed and a dual programmable CNN chip architecture is shown (combining analog array dynamics with logic). Finally, the CNN program design, i.e. analog template design methods, are summarized and the main features of a CNN workstation is presented.

1. Introduction

The 3–dimensional regular, analog, nonlinear, dynamic, locally connected processor array, called cellular neural network (CNN), has been invented in 1988 in Berkeley. In an unprecedented speed this paradigm has been spread out world wide: in two years almost hundred researchers made significant contributions most of them published in the Proceedings of the (first) IEEE Workshop on Cellular Neural Networks and their Applications CNNA–90 [19]. It can be considered as the 3D analog alternative of the 2D cellular logic automaton invented by John von Neumann.

The CNN is composed of *identical*, relatively simple, analog, *nonlinear dynamic units* (processing units, PU), these units are placed on a regular 3D geometric grid (several 2D layers), and the analog *interactions* between the units are local (within a finite neighborhood, a 3D window). The interactions between the units are represented by simple interconnections, although, they may be nonlinear and delay–type as well. Here, the term "neural" is just characterizing the few types of uniform units and the large number of interactions.

The CNN paradigm

- is a simple and powerful model for spatial dynamics in 3 dimensions
- represented by a set of dynamic spatial convolution equations on a finite spatial grid, thus,
- it is the most general deterministic representation of spatio–temporal phenomena (described by nonlinear and delay–type partial differential equations); moreover,

P. Dewilde and J. Vandewalle (eds.), Computer Systems and Software Engineering, 151–167.
© 1992 *Kluwer Academic Publishers.*

- the processing units (PUs) are in a one–to–one correspondence with the geometry (topology) of a sensory array (the processor indices are coordinates in the space, not just serial numbers);
- at the same time the CNN is a prototype programmable machine, the wealth of different problems can be solved using the same machine, and
- due to the local interconnectivity the VLSI realizations are dense, standard, and have unprecedented high speed (the first tested chips have a capability of 0.3 tera XPS per chip using a conservative technology with a simple multi–chip module possibility), and optical realizations are also in experimental phase capable of processing feed–forward templates with the speed of light.

Extending the CNN paradigm with local (global) logic (without any A/D or D/A converters!) the so called dual CNN is combining the strength of array dynamics and logic.

Starting results indicate that, using simple processing units only, the rich dynamics could yield very complex, yet useful, effects. Array oscillators, chaos generators, time varying interconnections are just a few of the many possibilities.

The fact that CNN processors are, at the same time, geometric objects provides natural representations of a lot of sensory array signals (tactile and visual perception, radiation, etc.).

CNN has extremely broad application potentials and wide spread impact in various scientific disciplines. Modelling complex mechanical, chemical, electromagnetic, geophysical and biological phenomena, e.g. stress analysis, corrosion patterns, transport dynamics, visual pathway models, pollution dynamics are just a few of the many disciplines and problems.

Programmability is a key property of the CNN paradigm. The representation of complex dynamics of thousands of interacting processing units with just a few "geometric" numbers of the cloning template is the basis of programmability.

To build a new generation of array processing computers with several orders of magnitude higher speed than present day supercomputers, this is a reasonable challenge to the CNN world.

Several starting results, case studies and projects on various fields already justify this broad scope of applications. Testing complex pattern errors in manufacturing processes, analysis of thermographic images, image processing including high speed motion analysis and camouflaged motion detection, robot control with optical signals, new models of important phenomena in the visual pathway are just a few of the many on–going projects world–wide.

In this paper the state of the art of this rapidly emerging field of computing is exposed by highlighting some key areas. After describing the general framework in Section 2, the programmability issue is considered in Section 3. Qualitative results are summarized in Section 4 (range of dynamics and stability). Section 5 shows application capabilities in three selected areas: motion detection, retina models, and layout error detection. In Section 6 the hardware realizations are reviewed and a dual CNN chip architecture is shown (combining analog array dynamics with local logic). Finally, the CNN "program design" i.e. analog template design methods are summarized and the architecture of a CNN workstation is presented in Section 7. Although, we tried to refer all major works in the areas selected, we are not claiming that the present list of references are complete.

2. The General Framework

The CNN is a locally connected, analog, dynamic, nonlinear processing array [1]. The processing elements are on a 2D grid (one layer) which can be multiplicated to have a multilayer CNN. One processing element (a simple cell) with nonlinear template elements [2] can be seen in Figure 1. Various grids are shown in Figure 2. The dynamics of the array can be described by the canonical equations (assuming $C_x = 1$, $R_x = 1$, $R_y = 1$)

$$\dot{v}_{xij}(t) = -v_{xij}(t) + \sum_{kl \in N_r(ij)} \hat{A}_{ij;kl}(v_{ykl}, v_{yij}) + \sum_{kl \in N_r(ij)} \hat{B}_{ij;kl}(v_{ukl}, v_{uij}) + I_{ij}$$

$$v_{yij} = f(v_{xij}); \qquad \text{(in many cases } I_{ij} = I\text{)}$$

where ij denotes a grid point associated with a cell on the 2D grid, kl is the grid point in the neighborhood of the ij cell and $N_r(ij)$ is the neighborhood of ij with a radius of r. $\hat{A}_{ij;kl}(v_{ykl}, v_{yij})$ and $\hat{B}_{ij;kl}(v_{ukl}, v_{uij})$ are the feedback and feedforward interconnection weights or template elements (atoms) uniquely defining the CNN array.

Delay template elements [2] add two additional terms:

$$\sum_{kl \in N_r(ij)} A^{\tau}_{ij;kl} \bullet v_{ykl}(t-\tau) + \sum_{kl \in N_r(ij)} B^{\tau}_{ij;kl} \bullet v_{ukl}(t-\tau)$$

In case of single variable, linear, space invariant templates the simple spatial convolution terms are

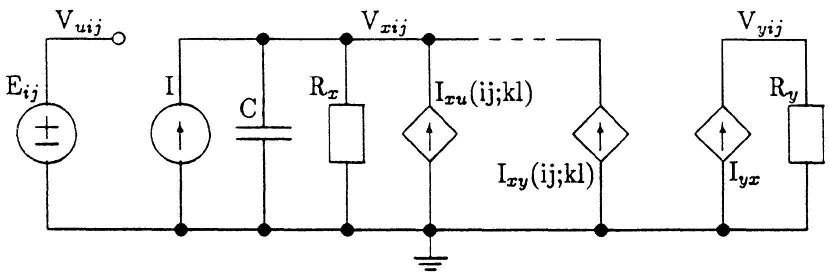
$$\sum_{kl \in N_r(ij)} A(ij;kl) \bullet v_{ykl}(t) + \sum_{kl \in N_r(ij)} B(ij;kl) \bullet v_{ukl}(t)$$

and the simple cloning template contains the $(r+1) * (r+1)$ real matrices A and B as well as the constant term I. In case of the nearest neighborhood ($r = 1$) these 19 numbers determine the task (the program) of the CNN array (see the details in [1] and [2]).

In case of array processing applications input signal arrays (e.g. grey-scale images) $S_1(ij)$, $S_2(ij)$ can be placed onto the CNN as

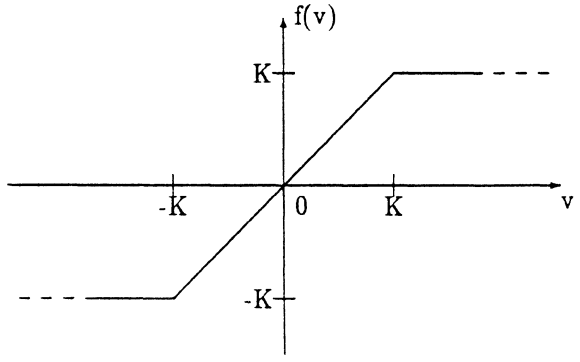
$$S_1(ij) = v_{uij} \text{ and } S_2(ij) = v_{xij}(0)$$

and the output signal array $S_o(ij)$ is associated with v_{yij}. The "program" is the cloning template. Normally, it is supposed that v_{uij} and v_{yij} are ≤ 1 in absolute values.



$$I_{xy}(ij;kl) = \hat{A}_{ij;kl}(v_{ykl}, v_{yij}) \qquad I_{xu}(ij;kl) = \hat{B}_{ij;kl}(v_{ukl}, v_{uij})$$

$$I_{yx} = f(v_{xij})$$



$$f(v) = \frac{1}{2}\left(|v + K| - |v - K|\right)$$

Fig. 1. The cell of a cellular neural network (nonlinear templates)

In very important applications these signal arrays are images and the CNN is used for solving complex image processing tasks. The CNN can be considered as a fully parallel real-time array processor with input signal arrays S_1 and S_2 and output signal array S_o. The computing time is the transient time or a finite time before the settling. This dynamics can be placed into a system setting too [3].

Hence the CNN has *two independent input signal arrays.* Continuous input, state, and output signal values (e.g. gray scale images) are represented by values $[-1, 1]$; (e.g. -1 is white, $+1$ is black and gray-scales are in between). If I_{ij} is not a constant it could mean a third independent input array.

Under some well defined conditions, the CNN with a single capacitor per cell and memoryless nonlinearity as well as with delays form the simplest form of a broad universe of capabilities [1–6]. An interesting second order cell CNN is the membrain-like one [5]. While the former can be interpreted as a kind of diffusion

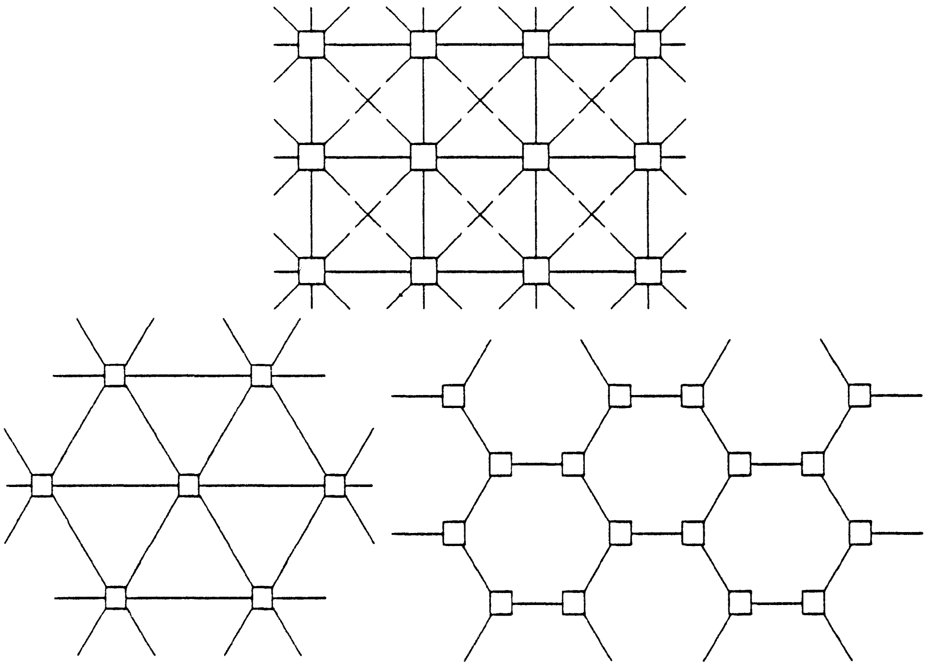


Fig. 2. Various cell–grids

equation the latter can be considered a wave equation for certain templates. Discretizing the state equation in time, numerical solutions of the dynamics of CNN can be calculated. In the same way, digital hardware accelerator boards can be designed.

3. Programmability

Exploiting the full capability of CNN the 2 (or 3) independent input possibilities can not be over–emphasized. The generic input S_1 can be time varying (continuous and discrete–time), the initial state S_2 can be changed in sampled mode only. The third input array is, in fact, a part of the cloning template: the constant term I. If it is not space invariant then, due to its cell–local nature, it can be used as another independent input array denoted by S_I. Its role in "spatial programming" will be discussed next.

The key of the programmability is the cloning template atom, or in circuit terms, the transconductance i.e. the voltage controlled current source (VCCS). It has the same role as the simple gate in digital systems. Hence, programming is based on this element. The cloning template is the elementary program ("instruction") of the CNN and this can be determined by giving its parameters, i.e. specific values and/or characteristics of the template elements (atoms). All these atoms can be realized by transconductances (VCCS) in case of analog VLSI implementation.

If the cell–local constant current term I (off–set current or threshold current) of the cloning template is space variant then we have an additional input array mentioned above. This array, which could be time varying as well, could have a role of a program. It is a *spatial program*. This, possibly time varying, spatial program could play a role of a reference map or a prescribed route etc. Since the dynamics of the CNN is very much depending on the value of this term (off–set current in VLSI circuits) it has a sensitive programming effect.

The non cell–local (*lateral*) part of the cloning templates can be programmed by changing their atoms (normally their values are space invariant). The central (cell–local) atoms (elements) of the A and B template matrices are fixed values in many applications.

The change of the cloning templates (in their template atoms) can be controlled continuously or within preselected discrete values. Before changing the cloning templates the CNN transient should be settled down. Since in many realizations the change of the template values can be performed much faster than the CNN array settling time, *time varying templates* can be realized as well.

Now, we have a very simple prototype programmable CNN machine. Beside the inputs and outputs, three basic notions are important:

- signal arrays Si,
- cloning templates TEMi (the programmable "instructions"), and
- analog "subroutines" SUBi, i.e. the parallel(P) and/or cascade(C) sequences of the templates. In case of parallel connections the cell–local operations on the parallel outputs are to be defined.

This way, CNN algorithms can uniquely and simply be defined [46]. Hence, algorithms like

```
VU:=S1;   VX:=S2;   S3:=TEM2(VU,VX);
SUB2=C(TEM1,TEM2);   S4:=SUB2(S3,0);
```

can be defined and executed.

In case of digital circuits two qualitative properties are needed for programmability:

- a prescribed range of dynamics and
- finite settling time (guaranteed stability).

In the next section these two properties are considered.

4. Range of Dynamics and Stability

4.1. Range of Dynamics

The range of dynamics is defined as the maximum absolute value of the state variables, $M \geq |v_{xij}(t)|$.

In [1] it has been shown that M is the sum of the absolute values of all the cloning template elements plus 1. In [2] it has been proved that similar condition is valid in case of nonlinear templates. Finally, in [14] the same condition was proved for delay–type templates as well.

Hence, given the allowable range of dynamics for a given technology, i.e. 5 or 15 Volts in a given CMOS VLSI design, then the allowable range of the cloning templates can be derived. Within this range the programmable CNN machine will calculate correct results (if it is stable).

4.2. Stability

Complete stability means that the CNN dynamics converges to a stable constant value for any input and initial state values. In the original paper [1] and in [16] the feedback (lateral) template symmetry has been defined as a sufficient condition, i.e. $A(ij;kl) = A(kl;ij)$, $kl =|= ij$. The positive cell–linking condition has been introduced in [11] as a sufficient condition which does not require the symmetry but the nonnegativity of the lateral feedback template elements (and the cell–linking property assures a kind of connectivity). In case of an important class of opposite–sign templates [11] not only some stability conditions were shown [12a] but chaotic behaviour was also found [12b]. The positive cell linking property was extended by stability preserving equivalent transformations [13] and the opposite sign template stability region has been extended as well [17]. Delay type template stability was considered first in [2] and extensive results were proved in [14] showing a set of simple conditions under which the delay is not disturbing the stability (e.g. for positive cell–linking templates). On the other hand, in [15] not only another important type of stability result has been found for delay–type templates in terms of also the delay value but an unstable symmetric delay–type template was found. The possibility of having multiple equilibria in CNN has been studied as well [18]. In the above stability results key mathematical theorems of [8–10] were used in many cases.

Thus, for a quite large class of templates stability can be assured, hence programmability within this class is allowed. It should be noted, however, that for some useful templates being "experimentally stable" no stability proof exists yet.

5. Selected Application Areas

To show the wide variety of problems solved by the same CNN prototype machine, three completely different application areas are described below.

5.1. Motion detection

The first motion detection results with CNN [23a] has been related to a famous experiment of Hubel and Wiesel. The problem is how to detect moving objects with a given speed and given direction. Next, following [23a and b] we will illustrate the results solving this kind of problems. The following cloning template is capable of detecting those and only those objects which have horizontal speed of one pixel per unit time. The consequtive images are applied in sampled mode. The present and last snapshots (S_2 and S_1) are placed onto the initial state and input of the cells, respectively, the detected image is the cell output signal array S_o. In fact, originally we have designed three consecutive templates performing set difference, speed detection, and direction detection. In this case it was possible to lump these three templates into a single template:

$$A = \begin{bmatrix} 0.0 & 0.5 & 0.0 \\ 1.0 & 2.1 & -2.0 \\ 0.0 & 0.5 & 0.0 \end{bmatrix}, \quad B = \begin{bmatrix} -0.5 & 0.5 & 0.0 \\ 2.0 & -8.0 & 0.0 \\ -0.5 & 0.5 & 0.0 \end{bmatrix}, \quad I = -11,$$

$$v_{uij} = S_1, \quad v_{xij}(0) = S_2, \quad v_{yij}(\infty) = S_o$$

Using this CNN, those objects moving horizontally with the specified speed will be seen on the output only, otherwise the output screen will be blank (white).

In the continuous mode of motion detection the continuously changing signal array $S(t)$ is placed onto the generic inputs of the cells, and the "present" vs. "next" sampling is made by a delay.

The continuously moving image $S(t)$ is placed on the CNN as v_{uij}. A template solving this problem is as follows:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0.25 & 0.25 & 0.25 \\ 0.25 & 2.00 & 0.25 \\ 0.25 & 0.25 & 0.25 \end{bmatrix}, \quad I = -4.75,$$

$$A^\tau = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad B^\tau = \begin{bmatrix} -0.25 & -0.25 & -0.25 \\ -0.25 & -2.00 & -0.25 \\ -0.25 & -0.25 & -0.25 \end{bmatrix}, \quad \tau \geq 10.$$

Another template solving the same problem (direction independent motion detection in continuous mode) is as follows:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad I = -2,$$

$$A^\tau = \begin{bmatrix} 0.68 & 0.68 & 0.68 \\ 0.68 & 0.68 & 0.68 \\ 0.68 & 0.68 & 0.68 \end{bmatrix}, \quad B^\tau = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \tau \geq 10.$$

Let us realize that structurally different templates can solve the same problem.

Sample input and output images are shown in Figure 3. Object B is moving faster, hence, it is disappearing.

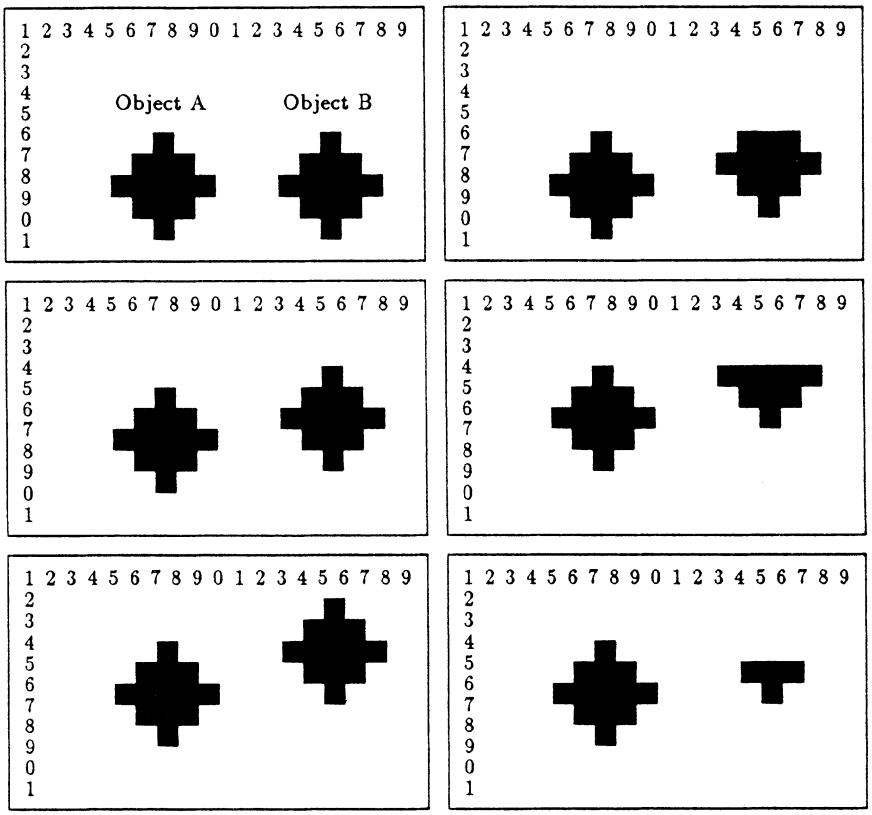


Fig. 3. Speed detection. Input snapshots (left) and output snapshots (right). The object moving with higher speed is disappearing in the output.

5.2. Retina models

The retina is one of the most understood part of living information processing systems. The success of the simple resistive grid model of the "silicon retina" [27] attracted deserved attention. As the resistive grid is a CNN with a special simple template, it was quite natural that more complex CNN templates were searched for finding more realistic retina models. Figure 4 shows a cross section of a CNN model. The 2D CNN array is supposed to be placed horizontally while the input to output parts are placed in the vertical direction. Thus the cell-local interactions (feedback or feedforward) are in the vertical direction, and the lateral interactions are connecting different cells. In detecting any motion in the visual pathway the structure [26] and a CNN model [29] is shown in Figure 5. The delay-type effects

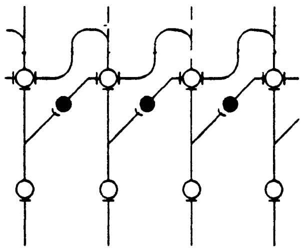


Fig. 4. The vertical cross section of a horizontally placed CNN layer resembling the retina structure

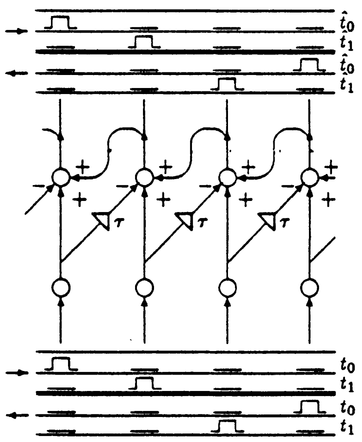


Fig. 5. Directionally sensitive nerve structure and CNN model

are important in many new visual models [e.g.28].

Hence, the CNN models are natural and simple models of vision–related biological systems having well–defined qualitative properties.

5.3. Layout error detection

Detecting layout errors in manufactured printed circuit boards (PCBs) and IC wafers are important problems ensuring production reliability. For PCB layout problems starting results [30] are clearly prove the feasibility of a system solution. The layout errors were reduced to the single problem of minimal line width detection. For rectilinear layouts the crucial detecting template for the vertical case, found by a new template learning algorithm, is as follows:

$$A = \begin{bmatrix} -0.1 & 0.4 & -0.1 \\ 0.4 & 0.0 & 0.4 \\ -0.1 & 0.4 & -0.1 \end{bmatrix}, \quad B = \begin{bmatrix} 0.2 & -3.0 & 0.2 \\ 0.0 & 2.5 & 0.0 \\ 0.2 & -3.0 & 0.2 \end{bmatrix}, \quad I = -5,$$

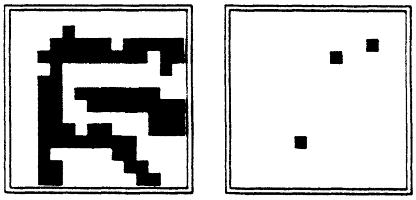


Fig. 6. Layout errors in the vertical direction

In Figure 6 an input layout (a) and an output (b) detecting violation places of a minimal wire width of 2 pixels in the vertical direction are shown.

6. Hardware Implementations

Since its invention CNN is tightly coupled with physical realization. The first attempts were VLSI circuits although an optical realization followed soon. A recent review [20b] is devoted entirely to this subject. The peek computing speed on silicon is now 0.3 Tera XPS on a 1 cm^2 area using a 2 micron technology (MOSIS) [38] and the programmable design [35] shows the capability of implementing CNN as a programmable prototype machine for performing general dynamic nonlinear spatial convolutions.

The deep relation between functional complexity and physical realization is an important aspect of designing modern electronic systems [31]. The VLSI implementation of general analog computing arrays, especially neural networks, now has a more or less established design technology [32]. The crucial element is the transconductance realizing the "synaptic" weight. The first commercially available "electronically trainable analog neural network" (ETANN) chip, the Intel 80170, is implementing programmable transconductances.

6.1. Fixed–template CNN

CNN is an appealing architecture for physical realization: local connectivity and regular repetitive geometric structure are both a "must" for success. Indeed, the various designs [33–39] exploited these architectural advantages. In practical terms it means that e.g using 10.000 transconductances for 160 inputs and *64 neurons in the fully connected* ETANN chip, more than 1000 neurons can be placed and interconnected on the same area in a locally connected CNN chip.

After the first design [33], the first fully tested high speed chip [38] proved the theoretical advantages. The 100 nsec time constant with 32 cells/mm^2 density using 2 micron technology is an impressive result which can be improved. The discrete–

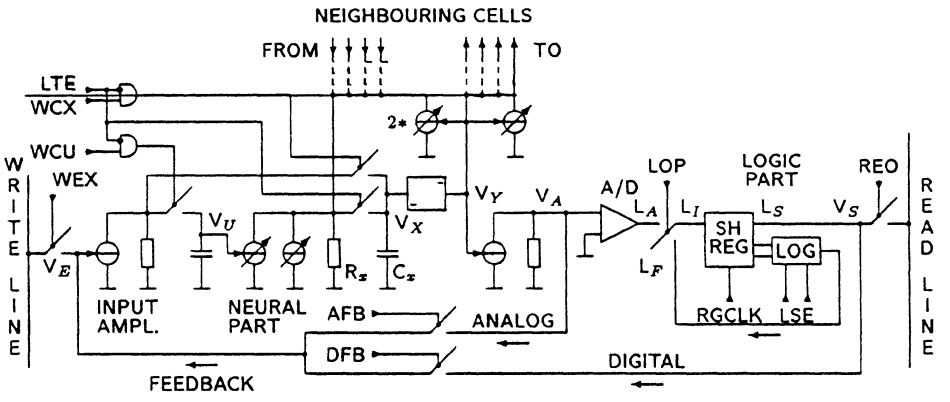


Fig. 7. The schematic of a programmable dual CNN cell

time CNN chip design [39] with 1200 cells on 1 cm^2 with a maximum speed of 3.3 MHz marks another important direction. The optical realization [37] provides speed of light processing using feedforward (B) templates only, the feedback (A) templates are implemented by electronic discrete–time feedback.

6.2. A programmable "dual" CNN chip

A key advantage of CNN processing is the locality of the interconnections. Once we transfer the signal array by wires, except for extracting the final results, we loose this advantage. To perform CNN algorithms, as defined in Section 3, without signal array transfers a new architecture and its VLSI implementation was introduced [34 and 35]. In this architecture, the first programmable CNN design, the sequence of CNN templates are performed by storing the intermediate results locally. Having this local analog storage, series and parallel connections of template sequences can be performed. Moreover, local logic is added as well. Hence, analog dynamics and logic operations can be combined realizing dual–type computing [7]. Unlike in hybrid computing there are no A/D or D/A converters, neither any digital (binary, decimal or hexadecimal) coding. The only coding is spatial.

The circuit schematic of a cell of this dual CNN chip containing logic, called CNNL [35], is shown in Figure 7. The programmable switches provide the correct timing of the prescribed template sequences. The local logical storage is capable of storing sequences of intermediate results, e.g sequence of processed image snapshots. The latter capability can conveniently be applied e.g. in achieving motion detection CNN algorithms. In the CNNL chip [35] the template element values can be programmed continuously and in discrete values controlled by standard logic. Using the CNNL architecture programmable CNN algorithms can be performed.

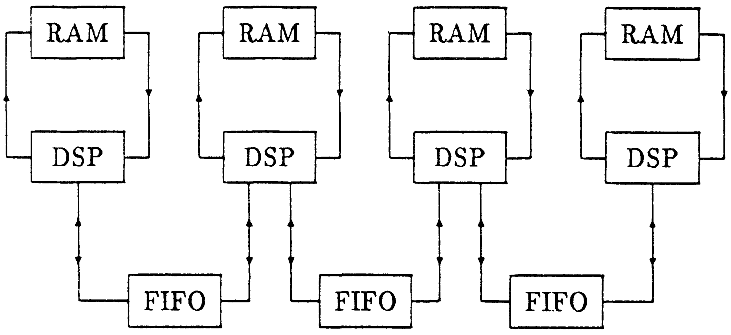


Fig. 8. The schematic of the CNN hardware accelerator CNN–HAC

6.3. A digital hardware accelerator

The local connectivity of the CNN processors was exploited in the design of a multiprocessor digital hardware accelerator CNN–HAC board [43]. In this design the half–million CNN cells were divided into 4 vertical strips, one DSP with local RAM storage was assigned to each strip and FIFO-s were used for communication between the DSP-s calculating the discrete–time representation of the CNN cell dynamics. The schematic architecture is shown in Figure 8. Using 4 Texas TMS320C26 DSP-s 2 microsec/cell/iteration was achieved.

Comparing the different CNN realizations the following computing time ranges are typical for the analog VLSI, digital hardware accelerator, and software simulator solutions, respectively, in the case of 1000 cell CNN-s: 0.1–1 microsec, 1–10 msec and 1–10 sec.

6.4. Some perspective realizations

Besides the optical implementations it seems reasonable that materials exhibiting spatio–temporal properties described by partial differential equations resembling the CNN dynamics in spatially discretized form could be used as an array computing device. The crucial question is the input and output. The image processing capabilities of some chemical phenomena [41,42] are promising initiatives. Living systems having locally connected regular geometric structures could serve as CNN prototypes as well.

7. Template Design and Development Tools

7.1. Various template design methods

Interestingly enough, quite a few useful templates were found by cut–and–try or heuristic methods.

Analytic methods considering some simple dynamic effects can also be used (see e.g. [40,25]). Using it with heuristics and experience a kind of personal design skill can be acquired.

Translating cellular automaton (CA), 2D filtering, and convolution–type image processing algorithms into CNN templates has been used as well (see e.g.in [22]). These areas are to be explored yet.

Learning procedures were proposed first in [48,49] and later in [30]. The convergence, the unimodality of the cost function, and the selection of the correct training set are crucial problems.

Different types of partial differential equations can be used in their spatially discretized forms to generate useful CNN templates. So called reaction diffusion equations [41] and autowave equations [42] are of special interest. The type of the equations is as follows:

$$\partial I(x,y,t)/\partial t = D \text{ divgrad } (g(I(x,y,t)) + F(I(x,y,t),L)$$

where $I(x,y,t)$ is the spatio–temporal intensity function, g and F are nonlinear functions, and D and L are constants.

Many sensory array processing biological models have the structure of CNN. The visual pathway is a typical case. It seems that the CNN model could be a unifying paradigm for modelling a lot of virtually different biological phenomena.

Methods combining known effects and designing CNN algorithms [46] from simple templates is are in their starting phase, although, interesting first results are already known (e.g. [23,25,30,40]).

7.2. A CNN workstation

For experimenting with various CNN templates a CNN simulator [44] has been developed and used in several Laboratories. Continuously developing it has led to the present form of a CNN Workstation [45]. This workstation has the following input signal array capabilities:

- optical scanners
- video camera with frame grabber
- interactive graphics
- ASCII files

Three simulators are integrated into the system:

- a general purpose software simulator
- a special purpose functional simulator for the dual CNNL chip
- a hardware accelerator board CNN–HAC

The workstation is a hardware–software integrated package which can be installed in any PC AT compatible computer.

Acknowledgements

This work has been supported by the research grant No. 2578/91 of the National Research Fund of Hungary (OTKA) and a joint grant No. INT 90-01336 of the National Science Foundation of the United States of America and the Hungarian Academy of Sciences. The inspiring discussions with Professor Leon O. Chua are gratefully acknowledged.

References

[1.a] L.O.Chua and L.Yang: 1988, 'Cellular neural networks: Theory', *IEEE Transactions on Circuits and Systems* **35**, 1257–1272

[1.b] L.O.Chua and L.Yang: 1988, 'Cellular neural networks: Applications', *IEEE Transactions on Circuits and Systems* **35**, 1273–1290

[2.a] T.Roska and L.O.Chua: 1990, 'Cellular neural networks with nonlinear and delay–type template elements', *Proc.IEEE CNNA-90*, 12–25

[2.b] T.Roska and L.O.Chua: 'Cellular neural networks with nonlinear and delay–type template elements and non–uniform grids', *Int.J.Circuit Theory and Applications*, to appear

[2.c] T.Roska and L.O.Chua: 1991, 'Dual CNN analog software', *Dual and Neural Computing Systems Laboratory, Computer and Automation Institute of the Hungarian Academy of Sciences (MTA SzTAKI), Budapest*, Report DNS-1-1992

[3] J.A.Nossek, G.Seiler, T.Roska and L.O.Chua: 1990, 'Cellular neural networks: Theory and circuit design', *Inst. Network Theory and Circuit Design, T.U.Munich*, and *Int.J. Circuit Theory and Applications*, Report No.TUM-LNS-TR-90-7, to appear

[4] H.Harrer and J.A.Nossek: 1990, 'Time discrete cellular neural networks: architecture, applications and realization', *Technical University of Munich*, Report No.TUM-LNS-TR-90-12

[5] J.Henseler and P.J.Braspenning: 'Membrain: a cellular neural network model based on a vibrating membrane', *Int.J.Circuit Theory and Applications*, to appear

[6] A.Radványi, T.Roska and K.Halonen: 1992, 'The CNNL architecture and time-varying templates', *Dual and Neural Computing Systems Laboratory, Computer and Automation Institute of the Hungarian Academy of Sciences (MTA SzTAKI), Budapest*, Report DNS-1-1992

[7] T.Roska: 1988, 'Analog events and a dual computing structure using analog and digital circuits and operators', in *Discrete Event Systems: Models and Applications* eds. P.Varaiya and A.B.Kurzhanski, Springer–Verlag, Berlin, pp.225–238

[8] M.W.Hirsch: 1985, 'Systems of differential equations that are competitive or cooperative II.: Convergence almost everywhere', *SIAM J.Math.Anal.* **16**, 423–439

[9] M.W.Hirsch: 1989, 'Convergent activation dynamics in continuous time networks', *Neural Networks* **2**, 331–349

[10] H.L.Smith: 1987, 'Monotone semiflows generated by functional differential equations.', *J.Diff.Eq* **66**, 120–442

[11] L.O.Chua and T.Roska: 1990, 'Stability of a class of nonreciprocal cellular neural networks', *IEEE Transactions on Circuits and Systems* **37**, 1520–1527

[12.a] F.Zou and J.A.Nossek: 1990, 'Stability of cellular neural networks with opposite sign templates', *Inst. Network Theory and Circuit Design, T.U.Munich*, Report No.TUM-LNS-TR-90-15

[12.b] F.Zou and J.A.Nossek: 1991, 'A chaotic attractor with cellular neural networks', *IEEE Trans. Circuits and Systems* **38**, 811–812

[13] L.O.Chua and C.W.Wu: 1991, 'The universe of stable CNN templates', *Memo. UCB/ERL, April, University of California at Berkeley*,

[14] T.Roska, C.W.Wu, M.Balsi and L.O.Chua: 1991, 'Stability of delay–type cellular neural networks', *Memo. UCB/ERL, November, University of California at Berkeley,*

[15] P.P.Civalleri, M.Gilli and L.Pandolfi: 1991, 'On stability of cellular neural networks with delay', *Politecnico di Torino*, Report, November

[16] G.Seiler and M.Hasler: 1991, 'Convergence of reciprocal cellular neural networks', *Inst. Network Theory and Circuit Design, T.U.Munich*, Report No.TUM-LNS-TR-91-12

[17] M.Balsi: 1991, 'Remarks on the stability and functionality of CNN's with one–dimensional templates', *Dual and Neural Computing Systems Laboratory, Computer and Automation Institute of the Hungarian Academy of Sciences (MTA SzTAKI), Budapest*, Report DNS-5-1991

[18] L.Vandenberghe and J.Vandewalle: 1990, 'Finding multiple equilibrium points of cellular neural networks without enumeration', *Proc IEEE CNNA–90*, 45–54

[19] 'Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications, CNNA–90', 1990, *Budapest*, IEEE Cat.No. 90TH0312-9

[20.a] T.Roska: 1991, 'Cellular neural networks: a state–of–the–art review', *Special Session on Cellular Neural Networks, Proc. European Conference on Circuit Theory and Design*, ECCTD–91, Copenhagen, pp.1–9

[20.b] T.Roska and P.Szolgay: 1991, 'A comparison of various cellular neural network (CNN) realizations — a review', *Proc.2nd Int. Conference on Microelectronics for Neural Networks*, 413–421

[21] T.Matsumoto et al: 1990, 'Several image processing examples by CNN', *Proc. IEEE CNNA–90*, 100–111

[22] L.O.Chua and B.Shi: 1989, 'Exploiting cellular automata in the design of cellular neural networks for binary image processing', *Memo.UCB/ERL M89/130, University of California at Berkeley,*

[23.a] T.Roska, T.Boros, P.Thiran and L.O.Chua: 1990, 'Detecting simple motion using Cellular Neural Networks', *Proc. IEEE CNNA–90* , 127–138

[23.b] T.Roska, T.Boros, A.Radványi, P.Thiran and L.O.Chua: 'Detecting moving and standing objects using cellular neural networks', *Int.J.Circuit Theory and Applications*, to appear

[24] V.Cimagalli: 1990, 'A neural network architecture for detecting moving objects II.', *Proc. IEEE CNNA–90*, 124–126

[25] T.Boros, K.Lotz, A.Radványi and T.Roska: 1990, 'Some useful new nonlinear and delay–type templates', *Dual and Neural Computing Systems Laboratory, Computer and Automation Institute of the Hungarian Academy of Sciences (MTA SzTAKI), Budapest,* Report DNS-1-1990

[26] A.M.Sillito and P.C.Murphy, 1988, 'GABAergic processes in the central visual system', in *Neurotransmitters and cortical functions (ed.R.W.Dykes and P.Gloor)*, Plenum Publ.Co., pp.167–185

[27] M.A.Mahowald and C.Mead: 1991, 'The silicon retina', *Scientific American* **264, No.5,** 76–82

[28] F.Wórgötter and C.Koch: 1991, 'A detailed model of the visual pathway in the cat: Comparison of afferent excitory and intracortical inhibitory connection schemes for orientation slectivity', *The J. Neuroscience* **11**, 1959–1979

[29] T.Roska, K.Lotz, J.Hámori, E.Lábos and J.Takács, et. al.: 1991, 'The CNN model in the visual pathway — Part I: The CNN–Retina and some direction– and length–selective mechanisms', *Dual and Neural Computing Systems Res.Lab., Comp. Aut. Inst., Hung.Acad.Sci., (MTA SzTAKI), Budapest*, Report DNS-10-1991

[30] P.Szolgay and T.Kozek: 1991, 'Optical detection of layout errors of printed circuit boards using learned CNN templates', *Dual and Neural Computing Systems Res.Lab., Comp. Aut. Inst., Hung.Acad.Sci.,(MTA SzTAKI), Budapest*, Report DNS-8-1991

[31] Á.Csurgay: 1983, 'Fundamental limits in large–scale circuit modelling', *Proc. European Conference on Circuit Theory and Design, ECCTD–83*, pp.454–457, VDE–Verlag, Berlin

[32.a] C.Mead: 1989, 'Analog VLSI and neural systems', *Addison Wesley*, Reading, MA

[32.b] C.Mead and M.Ismail (eds.): 1989, 'Analog VLSI implementation of neural systems', *Kluwer Academic Publ.*, Boston

[33] L.Yang, L.O.Chua and K.R.Krieg: 1990, 'VLSI Implementation of Cellular Neural Networks', *Proc. IEEE ISCAS-90*, 2425–27

[34] T.Roska: 1991, 'Dual computing structures containing analog cellular neural networks and digital decision units', *Proc. IFIP Workshop on Silicon Architectures for Neural Nets, Nice*, North Holland, Amsterdam, pp.233–244,

[35] K.Halonen, V.Porra, T.Roska and L.O.Chua: 1990, 'VLSI Implementation of a reconfigurable CNN containing local logic', *Proc. IEEE CNNA-90*, 206–215

[36] J.E.Varrientos, J.Ramírez–Angulo and E.Sánchez–Sinencio: 1990, 'Cellular neural networks implementation: a current–mode approach', *Proc. IEEE CNNA-90*, 216–225

[37] N.Frühauf and E.Lüder: 1990, 'Realization of CNNs by optical parallel processing with spatial light valves', *Proc. IEEE CNNA-90*, 281–290

[38] J.M.Cruz and L.O.Chua: 1991, 'A CNN chip for connected component detection', *IEEE Trans. Circuits and Systems* **38**, 812–817

[39] H. Harrer, J.A.Nossek and R.Stelzl: 1991, 'An analog implementation of discrete–time cellular neural networks', *Inst. Network Theory and Circuit Design, T.U.Munich, Munich*, Report No.TUM-LNS-TR-91-14

[40] L.O.Chua and P.Thiran: 1991, 'An Analytic method for designing simple cellular neural networks', *IEEE Trans. Circuits and Systems* **38**, 1332–1341

[41] V.I.Krinsky, V.N.Biktashev and I.R.Efimov: 1991, 'Autowave principles for parallel image processing', *Physica D* **49**, 247–253

[42] C.B.Price, P.Wambacq and A.Osterlinck: 1990, 'Image enhancement and analysis with reaction–diffusion paradigm', *IEE Proceedings* **137**, **Pt.I**, 136–145

[43] T.Roska, G.Bártfai, P.Szolgay, T.Sziranyi, A.Radványi, T.Kozek, Zs.Ugray and Á.Zarándy: 1990, 'A digital multiprocessor hardware accelerator board for cellular neural networks: CNN-HAC', *Int.J. Circuit Theory and Applications*, to appear
(earlier version: Proc. IEEE CNNA-90, pp.160–168)

[44] CNND simulator, Cellular Neural Network embedded in a simple Dual computing structure, User's guide Version 3.0, (ed.T.Roska and A.Radványi), 1990, *Computer and Automation Institute, Hungarian Academy of Sciences (MTA SzTAKI), Budapest*, Report No.37/1990

[45] A.Radványi and T.Roska: 1991, 'The CNN Workstation — CNND Version 4.1', *Dual and Neural Computing Systems Res.Lab., Comp. Aut. Inst., Hung. Acad. Sci., (MTA SzTAKI), Budapest*, Report DNS-12-1991

[46] T.Roska and L.O.Chua: 1992, 'The dual CNN analog software', *Dual and Neural Computing Systems Res.Lab., Comp. Aut. Inst., Hung.Acad.Sci., (MTA SzTAKI)*, Report DNS-2-1992

[47] Intel 80170 XA Electrically Trainable Analog Neural Network, 1990, *Advance Information*, Intel Corp.

[48] F.Zou, S.Schwarz and J.A.Nossek: 1990, 'Cellular neural network design using a learning algorithm', *Proc. IEEE CNNA-90* , 73–81

[49] S.Schwarz and W.Mathis: 1991, 'A design algorithm for cellular neural networks', *Proc.2nd Int.Conf. Microelectronics for Neural Networks*, 53–59

DEVELOPMENTS IN PARALLEL PROGRAMMING LANGUAGES

R. H. Perrott

Department of Computer Science

The Queen's University

Belfast BT7 1NN

N. Ireland

EMail: r.perrott@uk.ac.qub

Abstract. Sequential computing benefited from the fact there was one underlying model of computation on which hardware, algorithm, and software developments were based, namely, the von Neumann model. In the case of parallel computing there is at the moment no single model of computation. As a result there have been several approaches for the development of both hardware and software. In the case of parallel software the choice of programming language is no longer confined to a single approach. The main division of these languages is into either imperative or declarative languages. The declarative group can be further divided into logic and functional languages while the imperative group consists of procedural and object oriented languages. All these languages offer different ways of capitalising upon the power of parallel machines. However, to date it is not clear if any one approach is better than any other as not enough experience has been accumulated. This paper reviews the state of the art in the various types of parallel programming languages with particular reference to developments in Europe.

1. Introduction

In the case of sequential computers the architectural model, the programming paradigms and the method of constructing algorithms all have a single objective. In the case of parallel computers there is no longer a single architectural model to represent parallelism but rather a variety of different parallel architectures.

The main issue affecting the architectural model is how to organise multiple processors to execute in parallel. One of the first models was that of an array processor - the SIMD model - where multiple processors execute the same instruction but on different data; the processors operate under the control of a single processor which broadcasts the instructions to be executed. Array processors are particularly suited to problems involving matrices and some impressive results have been achieved. However, the main criticism of this model is that there is little flexibility in the architecture for problems which could benefit from the execution of different instructions at the same time - the MIMD model.

The earliest MIMD models were based on the shared memory concept where all the processors are connected to the same memory. In this scenario the processors can execute different parts of an application concurrently, thus hopefully reducing the time to execute the complete program. However, this model can lead to severe memory contention problems as the processors attempt to access the same data. There is some question as to whether this model will scale to larger orders of parallelism.

A more recent MIMD model is the distributed memory model where each processor has its own local memory. Processors communicate by passing messages. However, there is an overhead associated with such communication, which in many instances can be substantial. The size of the overhead is influenced by such factors as the distance between the two processors wishing to communicate and the interconnection topology. The distributed model is scaleable to greater orders of parallelism than that currently implemented.

In the case of parallel software the choice of programming language is no longer confined to a single approach. The main division of these languages is into either imperative or declarative languages. The declarative group can be further divided into logic and functional languages while the imperative group consist of procedural and object oriented languages (see later). All the various languages that have been proposed offer some different way of capitalising on the power of parallel machines. To date it is not clear if any one approach is substantially better than any other as enough experience has not yet been accumulated. In many cases the concepts have not been efficiently implemented on parallel machines. In addition there is a considerable lack of tools to assist in all aspects of parallel programming and debugging.

The third important aspect of programming parallel systems is the choice of algorithm. Studies have shown that transfering an efficient sequential algorithm to a parallel machine results in an inefficient parallel algorithm. It is now apparent that the design and construction of a new parallel algorithm for a particular application area can produce major performance improvements.

Hence, in the case of parallel systems there are three important and contributing factors, namely, the architectural model, the programming language and the choice of algorithm. The following sections concentrate on the programming language.

2. Approaches to parallel programming

Essentially there has been three main methods used to promote the wider use of parallel processing, namely

(i) extend an existing sequential language with features to represent parallelism.

The advantage of extensions is that existing software can be transferred to the new parallel machine with relative ease. This is possible because programmers are already trained in the base language and can introduce the extensions gradually as they become more familiar with the situation in which they should be used and the effect they produce. However, experience to date has shown that extension languages have been limited to a certain range of hardware and to machines with a small number of processors; there is some doubt as to whether this approach can be used on machines with a large number of processors. Problems have also been reported in the debugging of programs written in such languages as the interaction of the sequential and the parallel features can give rise to difficulties in detecting errors. A more general problem is that many of these extensions have been developed by different groups using the same language base which has led to non standard variants of the same language being produced making the production of a standard for such languages difficult.

(ii) implicit - use a sequential language but rely on the compiler to detect which parts of the program can be executed in parallel.

Most of the work in this area is based on Fortran and examines the DO loops of the program to determine if it is possible to spread the iterations of the loop across different processors.

The advantage of such an approach is that existing sequential programs can be moved relatively inexpensively and quickly to the parallel machine. This can represent a substantial saving in development costs and is an attractive proposition for many purchasers of a new parallel machine. However, it is rare that the full parallelism of the program is exploited without the help of a programmer to restructure the program; this usually requires a reorganisation of the loops of the program so that the automatic detection techniques will work.

In the case of the construction of new programs it is advisable that a programmer has some knowledge of the detection techniques if as much parallelism as possible is to be detected. This represents a diversion for a programmer from the main task of program construction. In addition, such an approach inhibits the development of parallel languages and algorithms as it is confined to a sequential notation.

(iii) develop a new parallel language.

In this case a completely new parallel language is developed ignoring all existing languages and applications. The main advantage of this approach is that a coherent approach to parallelism is presented. The parallel notation will enable a user to express directly the parallelism in an application and, in addition, will assist with the development of new parallel algorithms. However, it does mean that a user will have to rebuild the entire software base in the new language which is a labour intensive, expensive and perhaps an error prone exercise. All existing applications are ignored which requires courage on the part of the management of large installations, particularly since many new languages have not had the property of longevity.

These three approaches have been widely used over the years, and the following sections highlight some of the main examples.

3. Types of parallel programming languages

Since the early 1960s several proposals have been made in order to classify machine architectures, this has resulted in a 'periodic table' of computer architectures. It has not proved as easy to do this for programming languages since their characteristics do not lend themselves to easy classification. For the purposes of this article the classification shown below is used.

IMPERATIVE LANGUAGES

<u>PROCEDURAL</u>
ARRAY AND VECTOR PROCESSING
MULTIPROCESSING/DISTRIBUTED PROCESSING

<u>OBJECT ORIENTED</u>

DECLARATIVE LANGUAGES

<u>LOGIC</u>
<u>FUNCTIONAL/APPLICATIVE</u>

3.1 IMPERATIVE LANGUAGES

The imperative languages have the longest history of any language group. It was this approach which was used to program the early von Neumann machines and which has been refined over the years. The main characteristic is that the language enables a user to specify a sequence of operations to perform the calculation; the sequence of operations changes the state of the machine hence the description as 'state oriented' languages. The central operation of these languages is that of assignment which enables any location in the machine to be changed.

Within this broad category there have been separate developments under the headings of procedural languages and object oriented languages.

3.1.1 PROCEDURAL LANGUAGES

The procedural approach in a language means that constructs are provided for modularising the program code; this capitalises upon the experience which has been gained from the design and implementation of sequential languages.

There is more than one architectural model which can support this type of language approach. The main models are the array/vector processor and multiprocessor/distributed systems.

(i) Implicit Approach

An approach which was in widespread use on the early parallel machines is known as the implicit or detection of parallelism approach. This is an attempt to make the transfer from sequential to parallel programming as easy as possible for a user by placing the burden of parallelism detection on the compiler. In this way a user does not have to learn a new language or get involved with the complexities of parallel programming. The majority of this work is Fortran based.

Compilers based on this philosophy were first introduced with the advent of vector processors. Initially such machines had to handle data parallelism involving pipelined processing only but now they have been extended to multiple processor based systems.

In general, the main strategy of a parallelisation system is to examine nested DO loops, with the objective of vectorising the innermost loop and parallelising the outermost loop (assuming hardware support). The methods rely on data dependence analysis techniques which determine the flow of data in a program. This, in turn, enables statements which can be executed in parallel, to be identified. Data dependence analysis is the cornerstone on which all automatic parallelism detection methods are built; the quality of a paralleliser is directly related to the quality of the dependence analyser. Currently data dependence analysis techniques are available for nested DO loops but have not yet been commercially applied to complete programs.

At present the challenge in the field of parallelisation is to perform whole program optimisation. This requires full interprocedural analysis (the tracking of data across procedure calls) to be performed on a user program. Once a compiler uses interprocedural information as a basis for compile time decisions, data dependencies between procedures in a program can be resolved. The systems developed at Rice University and IBM Thomas J. Watson Research Laboratory provide a limited form of interprocedural analysis.

There are certain parallel programming situations which can be automatically parallelised without any user intervention. The most straightforward situation consists of loops with no data dependency between the iterations. In this case the iterations can be assigned to the processors either individually or in groups depending on the scheduling algorithm used by the particular system; in some systems a programmer has the option of specifying which processors to utilise.

In other situations if there is a possibility of a data dependency occurring within a program loop the compiler takes a conservative view which usually means that no parallelisation is attempted. The burden is then placed on a programmer to decide if the compiler's decision should be overridden. This is particularly the case in situations where interprocedural analysis is required as most existing systems are not capable of performing this analysis. Interprocedural dependencies are often difficult to detect and can require a considerable level of skill on behalf of a programmer.

Most of the work in this area has been based on the shared memory model. These systems have taken a considerable number of years to develop and are limited in what they can do.

The success of vectorisation in the 1970s has raised the expectations of the scientific programming community so that they now expect (if not demand) compilers which will automatically translate a sequential program for efficient execution on a parallel computer. This has meant that users are less tolerant towards new languages or language extensions.

The main research laboratories which have been active in this area are the following:

Cray Research with CFT (Minnesota);
University of Illinois with Parafrase (Illinois);
IBM with Ptran (New York);
Rice University with the PFC systems (Texas);
Superb as part of the Suprenum project (Europe);
Velour at Honeywell Bull (Europe).

In addition many computer manufactures such as Convex and Alliant have produced systems for their particular machine.

(ii) Explicit Approach - Array and Vector Processing Languages

Languages for array and vector processor machines should provide features which enable a programmer to express the parallelism in the data of the application as well as its subsequent manipulation. This type of parallelism is therefore data oriented and, as a result, it is sometimes referred to as lockstep parallelism; a single instruction is broadcast and applied simultaneously to many different data sets.

This approach has been particularly widely promoted by array processors such as the Illiac IV (Barnes 1968, Hord 1984), AMT Distributed Array Processor (Parkinson, 1983), Connection Machine (Hillis, 1985) etc.

Since the early days of these machines, particularly in the case of array processors, languages have been designed and implemented which have included features to represent data parallelism. The main method of achieving this is to use the array data structure and to enhance its syntax to include parallelism. Most of the work in this area has been carried out by computer manufacturers and involves the manufacturer's particular architecture. Another characteristic of these languages is that the majority of the software developed is based on Fortran; this is hardly surprising since the targetted users are engineers and scientists.

Typical of these languages are CFD Fortran for the Illiac IV (Stevens, 1974; Vectran at IBM (Paul and Wilson, 1975); Fortran Plus for the AMT Distributed Array Processor. Exceptions to this Fortran trend are the languages Actus (Perrott, 1979) which is Pascal based and the language Booster (Paalvast and Sips, 1989). The first two projects were carried out in the USA, the last three in Europe.

In this area of parallel programming the proposed successor to Fortran 77, currently referred to as Fortran 90, will probably have most impact. This next Fortran standard will contain features to enable the specification of data parallelism and its manipulation. This proposed Standard incorporates some of the ideas that have been proposed in the earlier mentioned languages.

If past experience is any guide, then Fortran 90 will be the dominant parallel programming language in this area as the majority of manufacturers will implement the standard, and the majority of users will prefer to use a standard programming language.

(iii) Explicit Approach - Multiprocessor/Distributed Systems

In the case of languages for multiple processor systems the imperative approach has received the most attention. This arose out of the work carried out on operating systems in the early 1960s. At that time programmers were designing programs to control and coordinate the many independent activities in an operating system. It is this work which laid many of the foundations for this type of parallel programming language.

The term **process,** or more recently **task,** is used to describe a sequence of program instructions that can be performed in parallel with other groups of program instructions. A program can therefore be represented as a number of processes which can be executing concurrently. The point at which a processor is withdrawn from one process and given to another is dependent on the progress of the processes and the algorithm used to assign the available processor(s). The nett effect is that processes are capable of interacting in a time dependent manner.

Thus, in a concurrent programming environment, a programmer requires not only program and data structures similar to those required in a sequential programming environment but also mechanisms to control the interaction of the processes - processes which are proceeding at fixed but unknown rates.

The situations in which processes interact can be divided into two categories. The first situation occurs whenever several processes wish to access a resource at the same time. For example, when several processes wish to update a shared variable, only one process must succeed in gaining access to the resource at any time. Once a process has obtained the resource it must be able to use the resource without interference from the other competing processes; this is referred as **mutual exclusion**.

The second situation occurs whenever processes are co-operating, they must be correctly synchronised with respect to each other's activities. For example, when one process requires a result not yet produced by another process, the first process must be able to wait on the second process and the second process must take the responsibility of resuming the first process when it arrives with the result. The processes are therefore scheduling one another and are aware of each other's existence and purpose; this is referred to as **conditional synchronisation**.

The methods which have been used for solving these problems have evolved over the years. Originally they were at a low level and involved instructions like test and set, where two operations could be performed without interruption. In the early 1960s Dijkstra introduced the semaphore based on the railway signalling system. Although sufficient to solve the problems they could lead to programming situations which were difficult to understand and complex in nature.

Experience and research produced a series of solutions such as critical regions and conditional critical regions, finally resulting in the monitor (Perrott, 1987). The monitor can be regarded as state-of-the-art in shared memory concepts for parallel programming. The monitor was influenced by the class concept of Simula.

A monitor defines a shared data structure and all the operations that can be performed on it. These operations are defined by the procedures or functions of the monitor. In addition, a monitor defines an initialisation operation that is executed when its data structure is created.

In general, a process can access the shared data of a monitor by calling one of the monitor's procedures. If there is more than one call then only one of the calling processes is allowed to succeed in entering the monitor at any time; this is to guarantee that the data of the monitor is accessed exclusively. Only when a process exits the monitor is it possible for one of the calling processes which was delayed to enter the monitor.

It is also possible for a process to enter the monitor and discover that the information it requires has not yet arrived. In such a situation, it can join a queue associated with that condition and thereby release its exclusive access over the monitor, after which another process is now able to enter the monitor. Eventually, a process may enter the monitor and enable a suspended process to continue. The queues within a monitor are usually identified by condition variables and a process can append itself to a single condition variable queue by executing a **wait** operation. Another process executing a **signal** operation on a condition variable queue will cause a process delayed on that queue (if there is one) to be resumed.

A second technique which can be used to solve the problems associated with mutual exclusion and conditional synchronisation is based on message passing.

In 1975 Dijkstra (Dijkstra, 1975) introduced, the concept of a guarded command. A guarded command is simply a guard, which is a boolean expression, followed by a list of statements. If the guard is true the associated statements can be executed otherwise they cannot. It is possible to group several of these guarded commands together to form selection and repetitive constructs.

It was Hoare (Hoare, 1978) who incorporated these ideas into the notation known as communicating sequential processes (CSP). The essential idea is that synchronisation is set up by message passing using input/output commands. Effectively an output command in a sender process must specify the destination process. An input command in a receiver process must specify the source process and the parameter lists must match. Only under these conditions is it possible for two processes to communicate. Hoare took this idea and combined it with Dijkstra's guarded commands by enabling an input command to occur in a guarded command. One significant feature of Hoare's processes is that they must name each other in order to communicate; there is a symmetrical relationship between the processes. This was felt to be restrictive and Brinch Hansen (Brinch Hansen, 1978) proposed a notation known as distributed processes (DP) in which the process which is called does not need to know the name of the process which called it. This would seem to be reasonable in an environment, for example, where a library facility was being called by many processes. It is not necessary for the library process to know which process is calling it. This amounts to an asymmetric relationship between the calling and called processes.

This technique of message passing has been adopted in languages such as Ada (Ada, 1983) and Occam (Inmos, 1984).

Hence, with this technique, all communication is explicit through the transfer of values; there is no reading and writing of shared variables and no explicit queues have to be manipulated by a user. A significant enhancement to this technique is the idea of non determinism, in other words given a choice of several true guards one is picked at random. This was meant to reflect the way in which real time events occur; it is not possible to predict their order and therefore this should be incorporated into any programming notation.

The disadvantages of using a new parallel language is that it requires the users to rebuild their entire software base - a highly labour intensive and error prone operation. By doing so it ignores all the existing applications which have been accumulated. In addition, there is strong resistance against a new language by some users - the not-invented-here syndrome. However, the major advantage of using a new language is that it provides a coherent approach to parallelism, including a parallel notation, in which to design and discover new algorithms.

As a result of the proliferation of parallel Fortran dialects, steps have been taken in the US to try to standardise parallel languages for shared memory machines. A number of US academics and suppliers of parallel machines have been meeting as a group known as the Parallel Computing Forum (PCF). The objective of this Forum is to produce common extended Fortran syntax and semantics which allow concurrent processing to be expressed easily. The membership of the Forum is such that its deliberations are likely to be widely adopted.

The PCF syntax and semantics provides basic parallel constructs for specifying parallel processing, suitable for execution on a shared memory multiprocessor. One goal of these extensions is to avoid a program's dependence on the actual number of processors available at any given time to execute the program. Since a programmer is not constrained to coding for a limited number of processors the program may describe either more or less parallelism than there are processors in the actual system. An implementation of these extensions requires at least one processor to execute a parallel construct. Consequently a programmer can control the number of processors executing a program.

Recently the deliberations of the Parallel Computing Forum have been passed to the US Standards organisation, ANSI, who have set up a committee X3H5 to produce a Fortran based standard for shared memory multiprocessor machines. If, or when, such a standard is produced it is likely to have a major impact on languages for multiprocessor machines throughout the US and beyond.

Although the experience that has been accumulated in the use of these languages is limited there does appear to be some consensus and guidelines emerging. It appears relatively easy to write programs for shared memory machines while debugging is difficult. This indicates that by using shared variables the effect of doing something incorrect can manifest itself at a later time, giving no indication of when the error occurred and proving extremely difficult to detect. Distributed memory systems appear to be hard to program but easier to debug. The easier debugging is a consequence of the data being distributed and therefore the effects of an error being localised. Data organisation is therefore the key to efficient programming even on shared memory multiprocessors. These results are based on the scientific users first experience with these machines and do not involve the use of languages of such sophistication as illustrated earlier.

3.1.2. OBJECT ORIENTED LANGUAGES

In the last decade an approach to imperative programming which has gained widespread popularity is that of object oriented programming. In this approach an object is used to integrate both data and the means of manipulating that data. Objects interact exclusively through message passing and the data contained in an object are visible only within the object itself. Objects are intended for structuring programs in a clear and understandable way. Thus the connection with the class structure of Simula is very strong.

The behaviour of an object is defined by its class, which comprises a list of operations that can be invoked by sending a message to the object. All objects must belong to a class. Objects in a class have the same properties and can be manipulated using similar operators. The definition of an object class can act as a template for creating instances of the class. Each instance has a unique identity but has the same set of data properties and responds to the same set of operators.

Inheritance, allows a class to be defined as an extension of another (previously defined) class. Typically when a new object class is created a place for it is defined within the class hierarchy. The effect of this is that the new class inherits the state and operators of its superclass in the hierarchy. In addition, new properties of the state and new methods may be defined to augment the new object class.

Communication between objects is achieved by the passing of messages. A message is a request from the sender for the receiver to execute a procedure, called a method. An object is activated when it receives a message from another object. While the receiver of the message is active, the sender is waiting for the result, so the sender is passive. After returning the result, the receiver becomes passive again and the sender continues. At any time only one object in the system is active.

Object oriented programming encourages the grouping together of pertinent information and enforces the encapsulation of this information according to an explicit interface. For users of a certain class, the set of available operations, together with a description of their behaviour is all that is relevant. The interior of the objects, the variables and the code, is completely inaccessible to them.

Most existing object oriented languages are sequential in nature and observe the following restrictions:

(i) execution starts with exactly one object being active,
(ii) whenever an object sends a message, it waits until the result of that message is returned,
(iii) an object is only active when it is executing a procedure in response to an incoming message.

At any moment there is exactly one active object, although very often control is transferred from one object to another.

There are several possibilities for the introduction of parallelism, namely,

(i) objects are active without having received a message;
(ii) receiving objects continue to execute after returning results;
(iii) messages are sent to several active objects at the same time;
(iv) senders of a messages proceed in parallel with the receivers.

These possibilities can be realised by associating a process with each object. The introduction of processes means that several processes can be active at the same time. Facilities which provide synchronisation and mutual exclusion are usually supplied as built-in classes, for example, the semaphore. Examples of languages which have followed this approach are Concurrent Smalltalk (Yokote et al, 1986), Emerald (Black et al, 1987) etc.

Allowing a sender to proceed in parallel with a receiver corresponds to asynchronous rather than synchronous message passing. With synchronous message passing the sender is blocked until the receiver has accepted the message. Thus, the sender and receiver not only exchange data but they also synchronise their activities. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept a message. Conceptually the sender continues immediately after sending the message. It is possible to obtain a large degree of parallelism after a number of messages have been sent. This scheme has been adopted most notably by the family of actor languages (Agha 1986).

Parallel object oriented programming languages have many similarities with the earlier parallel procedural languages. For example, in using processes to identify a section of code which can be executed in parallel and providing facilities to solve the problems of mutual exclusion and synchronisation. However, their major advantages are in the areas of data hiding and code reuse.

3.2 DECLARATIVE LANGUAGES

Another major approach to language design is that represented by the declarative approach. In this case a notation is used to represent what is to be calculated rather than how the calculation is to be performed. How the calculation is performed depends on the implementation of the language and is not meant to be a concern of a programmer. A major difference between this approach and the imperative approach is that in the declarative approach there is no concept of state; a programmer does not work or think in terms of state transitions.

Essentially there are two main models

(i) the logic model which is based on the relation between entities;
(ii) the functional model which is based on the use of functions.

The main argument in favour of the declarative approach compared to the imperative approach is that the notation is at a higher level of abstraction so that a user does not, for example, have to be concerned about architectural details.

An attractive property, particularly for computer scientists, is that formal assessment of declarative languages is easier due to the their mathematical basis; the ability to prove a program correct is becoming an increasingly important property. However, a major criticism is that declarative languages have not been efficiently implemented; the advent of parallel architectures represents an opportunity to explore their efficient implementation.

3.2.1. LOGIC LANGUAGES

In this type of programming the underlying computational unit is the relation and the approach concentrates on proving entities rather than building entities as in imperative languages. A logic program is constructed as a finite set of clauses where a clause is a formula of the form

$$H \leftarrow B1, B2, \text{---}\ Bn \qquad n >= 0$$

where H is referred to as the clause's head and B1, B2, --- Bn its body.

It can be thought of as either (i) a logical statement, that is, if all the B's are true then H is true - the declarative interpretation.or (ii) to prove theorem H you have to prove sub theorems B1, B2 etc. - the procedural interpretation.

The earliest programming language based on this approach is Prolog (PROgramming in LOGic) which was developed in the early 1970s by Colmerauer and his colleagues at the University of Marseilles. It has become an essential language in the artificial intelligence field, although it has been used in many other application areas.

Programming in Prolog involves the specification of a set of facts and rules to form a database of knowledge on a subject. Questions may be asked of this knowledge database via Prolog which is then responsible for making inferences. In practice, a Prolog program consists of a set of clauses where a clause can be a fact, a rule or a question. Prolog uses Horn clauses which are clauses having at most one literal (one then) in their head part.

The basic data manipulation operation in logic programs is unification which results in a substitution. Operationally a substitution can be thought of as a simultaneous assignment of values to variables except that a variable can only be assigned a value once, and the value assigned can itself be another variable or a term containing variables. Hence logic variables have the single assignment property; initially they are unbound but once they receive a value, by unification, they cannot be changed.

Logic languages, like Prolog, proceed backward from a goal and use a depth first, left to right search to try to find answers that satisfy the query. When a dead end is encountered the implementation backtracks, that is, retraces its steps and then goes down another path. This sequential behaviour is a consequence of having to execute on a sequential processor. In order to improve efficiency in many implementations extra logical control features have been introduced.

There are several opportunities for introducing parallelism. The main sources of parallelism can be classified as

(i) AND-parallelism comes from trying to solve many parts of the problem simultaneously, that is, pursuing solutions to a number of subgoals at the same time. The main goal is solved if the first subgoal is solved AND, the second AND, the third etc.

(ii) OR-parallelism involves the evaluation of a problem in several ways at the same time. The problem is solved if the first clause is solved OR, the second OR, the third etc.

In summary, OR-parallelism involves evaluating statements in parallel and this can be facilitated by taking a copy of the variables involved. AND-parallelism involves working on the body of each statement in parallel.

In concurrent logic languages backtracking is difficult since bindings would have to be undone and all processes that used the binding backtracked. As a result, backtracking is avoided in concurrent logic languages, all clauses are searched in parallel and no bindings are allowed during parallel executions to be visible outside until one of the parallel executions is committed to. This can lead to exponential growth which can be controlled by non deterministic selection.

3.2.2. FUNCTIONAL LANGUAGES

Functional programming, as the name implies, involves the use of functions as the basis for the construction of programs. A functional program comprises a set of equations describing functions and data structures which a user wishes to compute. The application of a function to its arguments is the only control structure; there are no features which deal with assignment, conditional or looping structures as in imperative languages. The application of a function to a variable can produce a new variable and a new value but never change an old value. Functions are regarded as in the mathematical sense in that they do not give rise to side effects that can occur in, for example, Fortran functions. As a consequence a value of a function is determined solely by the values of its arguments, a property which is referred to as referential transparency; side effects mean the loss of referential transparency. In this approach to programming there is no concept of state which is associated with imperative languages.

It is claimed that, in comparison to imperative languages, functional languages have simpler semantics and are easier to write, understand, manipulate as well as manipulate in program transformations and correctness proofs. One of the major goals of the functional language community is to demonstrate that side effect free programming can be achieved without sacrificing efficiency or modularity; parallel architectures may enable this goal to be realised.

The introduction of functional programming was heavily influenced by the lambda calculus. In the lambda calculus the term XY means X applied to Y while λX.(X+1) is an expression for the function f(X) = X+1. The lambda indicates that a function is being defined where X is the bound variable of the function, and .X+1 is the body or form of the function. The only execution step possible is the application of a function to its argument. The function is on the left and its argument is the next complete expression to the right. The argument should be substituted where occurrences of the function's bound variable appear in the function body.

The earliest language which utilises this approach was LISP (McCarthy, 1960, 1963). LISP is based on function evaluation but employs imperative constructs to control the process of expression evaluation. The list is the essential data structure. There are good function abstraction features but poor data abstraction features. Modern variants, for example, Scheme (Rees and Clinger, 1986), have introduced stronger type checking, and also scoping of names to help in the production of modular software.

The main features which characterise the more recent functional languages are the following :

(i) higher order functions
means that functions are treated as first class values which can be stored in data structures, passed as arguments and returned as results. Thus, functions are values just like any other values in the program and can appear anywhere a value can appear.

(ii) lazy evaluation (or call by need)
means that in a functional expression the arguments are reduced only on demand, that is, the arguments to a function are evaluated only when needed and not when the function is called, so unneeded results are never calculated. The opposite technique is eager (greedy) evaluation where the arguments to a function are always evaluated before the function is applied. Lazy evaluation has been advocated as an optimal evaluation strategy for the implementation of functional programs on a sequential processor, optimal that is with respect to evaluation time, however, call by need is not optimal with respect to space.

(iii) equations and pattern matching
means programmers can write several equations when defining the same function; only one of the equations is applicable in a given solution. Pattern matching is used to determine which equation is applicable; this is referred to as equational reasoning in the design and construction of programs.

(iv) data abstraction
improves the modularity, security and clarity of programs. Strong static checking helps in reducing errors. This is true for other language approaches as well. An infinite data structure is equivalent to a recursively defined data structure.

Functions which return defined results even when the arguments are undefined are called non strict whereas functions which are always undefined with undefined arguments are called strict. In the latter case a computation can have a well defined outcome even though some of its component computations may be undefined, provided they are not needed to define that outcome. Lazy evaluation is compatible with a non strict interpretation of a function definition, while eager evaluation is compatible with strict interpretation of a function definition.

Recently a group of research workers in the field of functional programming got together to produce a standard for functional languages, one of their concerns was to avoid a proliferation of languages in this area. The result of their effort is a new functional programming language called Haskell (Hudak and Wadler, 1988). The standardisation process was an international effort and involved research workers from both Europe and America. Haskell is a purely functional programming language with higher order functions, lazy evaluation, static polymorphic typing, user defined data types, and pattern matching. Haskell also has a module facility, a well defined functional input/output system and a rich set of primitive data types including lists, arrays, arbitrary and fixed precision integers and floating point numbers.

In general, there are still inefficiencies connected with the implementation of features such as higher order functions and lazy evaluation; these are concerned with the overhead of dynamic storage management. Also the removal of side effects results in the need for more memory capacity and bandwidth which produces a bottleneck in a sequential von Neumann machine. Many of these problems must be solved before functional languages can execute with speeds comparable to that of imperative languages.

The benefits of parallel processing for functional programming are realised by means of the Church Rosser Theorem which states that any reduction sequence which yields a reduced form may be used in the evaluation of a function. More specifically, if a given lambda expression is reduced by two different reduction techniques and if both reduction sequences yield a reduced form then the reduced forms are equivalent up to renaming of bound variables.

Informally this property states that the same answer is produced when reducing any expression, no matter in what order the indexes are reduced provided the reduction sequence terminates. If the evaluation sequence does not matter then the reduction can be done in parallel without worrying about the order in which they finish.

If functions do not have side effects then it makes no difference in which order they are executed and it is therefore possible to evaluate them in parallel. The only delay may be that a function must wait on a result being produced by another function. The compiler should analyse the program to decide on the evaluation order.

Parallelism in functional languages is therefore manifest through data dependencies and the semantics of primitive operators in contrast to conventional languages where explicit constructs are typically used to invoke synchronisataion and coordinate concurrent activities. In most systems parallelism is detected by the system and allocated to the processors automatically - implicit parallelism.

Hudak (Hudak, 1989) argues that functional programming is just an evolution of the imperative approach since languages like Fortran had expressions as an important feature with a mathematical flavour and subsequent languages like Pascal built on this theme using the function. Functional programming is therefore the logical conclusion since everything is treated as an expression.

4. Summary

A complete solution to the problem of parallel computing involves several components: the design of parallel algorithms, the type of programming language and the characteristics of the underlying architecture. Currently it is the programming languages (and tools) which require the most urgent attention.

In the imperative approach explicit parallel programming features are provided to enable a programmer to construct a parallel solution and to help the implementation utilise the parallel processing capability of the underlying architecture. In the declarative approach the parallelism is not (normally) explicit in the language; and the burden is placed on the implementation to realise the parallelism in the solution. In certain projects each approach has 'borrowed' features from the other approach. For example, there are declarative languages which provide explicit parallel features to assist a user in identifying the parallelism in a solution. Conversely, in the imperative approach there has been much research into automatic parallelisation of sequential programs.

Recently it has been pointed out that the differences between these two approaches may not be as great as originally thought. The reasoning is as follows: in languages, like Fortran, the expression is an important concept with a strong mathematical basis; subsequent languages, like Pascal, built on this theme using the function. Functional programming can therefore be regarded as the logical extension of this trend where everything is a function/expression.

References

Ada 1983. Reference Manual for the Ada Programming Language. (ANSI/MIL-STD 1815). United States Department of Defense, Washington, D.C.

Agha G. 1986. Actors : A model of concurrent computation in distributed systems. MIT Press.

Barnes G H., Brown N.M., Kato M., Kuck D., Slotnick D., and Stokes N A. 1968. 'The Illiac IV Computer'. IEEE. Trans. on Computers, 17, 746-757.

Black A., Hutchinson N., Jul E., Levy H. and Carter L. 1987. 'Distribution and abstract types in Emerald', IEEE Trans Software Eng SE 13, 1 (Jan). 65-76

Brinch Hansen P. 1978. 'Distributed Processes: A Concurrent Programming Concept'. Communications ACM, 21, 934-940.

Dijkstra E W. 1975. 'Guarded Commands, Non-determinacy, and Formal Derivation of Programs'. Communications ACM, 18, 453-457.

Hillis W D. 1985. The Connection Machine, MIT Press.

Hoare C A R. 1978. 'Communicating Sequential Processes'. Communications ACM, 21, 666-677.

Hord R M. 1984. The Illiac IV - The First Supercomputer. Computer Science Press, Maryland.

Hudak P. 1989. 'Conception, evaluation, and Application of Functional Programming Languages'. ACM Surveys, 21, 3, 359-411.

Hudak P., and Wadler P. Eds. 1988. Report on the Functional Programming Language Haskell. Tech. Rep. YALE U./DCS/RR656. Department of Computer Science, Yale University.

Inmos Ltd. 1984. Occam Programming Manual. Prentice-Hall, Englewood Cliffs, New Jersey.

McCarthy J. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. Communications ACM 3,4, 184-195.

McCarthy J. 1963. A basis for a mathematical theory of computation. In Computer Programming and Formal Systems. North-Holland, The Netherlands, 33-70.

Parkinson D. 1983. 'The Distributed Array Processor'. Computer Physics Communications ACM, 28, 325-336.

Paalvast E M. and Sips H J. 1989. 'A high level language for the description of parallel algorithms', Parallel Computing 89 Published by North Holland.

Paul G. and Wilson W. 1975. The Vectran language: and experimental language for vector/matrix array processing' IBM Palo Alto Scientific Center, Report 6320-3334.

Perrott R H. 1979. 'A Language for Array and Vector Processors'. ACM Trans. on Programming. Lang. Systems, 2, 177-195.

Perrott R H. 1987. Parallel Programming. Addison-Wesley.

Rees J. and Clinger W. Eds. 1986. The revised report on the algorithmic language Scheme. Sigplan Notices 21, 12, 37-79.

Stevens K. 1974. 'CFD - A FORTRAN-like Language for the ILLIAC IV'. Sigplan Notices, 10, 72-80.

Yokote Y. and Tokoro M. 1986. 'The design and implementation of Concurrent Smalltalk'. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. Sigplan Not (ACM), 21, 11, 331-340.

LOAD BALANCING GRID-ORIENTED APPLICATIONS ON DISTRIBUTED MEMORY PARALLEL COMPUTERS

D. ROOSE, J. DE KEYSER and R. VAN DRIESSCHE
Katholieke Universiteit Leuven
Dept. of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee-Leuven, Belgium

Abstract. Load balancing grid-oriented applications on distributed memory parallel computers requires an optimal partitioning and distribution of the data among the processors. In this paper, we first summarise typical properties of grid-oriented applications by describing the types of grids that often arise in the solution of partial differential equations, together with characteristics of iterative algorithms and the resulting data dependencies. We survey some important methods for partitioning and mapping that take into account these properties. Two techniques are discussed in more detail : recursive bisection based on a cost function and genetic algorithms.

Key words: parallel computing, distributed memory, load balancing, graph partitioning, task-to-processor mapping problem, grid-oriented applications

1. Introduction

Distributed memory parallel computers consist of a number of nodes, i.e. processors with a private local memory, connected by a communication network. Parallel computers with distributed memory are gaining importance and are now surpassing vector processors and shared memory multiprocessors in performance.

At present, the principles of parallel algorithm design are well established. Most often, the data set and the associated work are partitioned and distributed among the processing nodes. This 'data parallelism' is especially well suited for grid-oriented problems, such as the numerical solution of partial differential equations. Nevertheless, the development of parallel algorithms and software is often difficult and tedious. This paper addresses one of the main reasons for this.

The problem of finding an appropriate *partitioning* and *mapping* (distribution) that minimises (approximately) the execution time is called the *load balancing problem.* Being an NP-complete problem, load balancing is inherently very difficult. In order to find a good solution one should exploit as much as possible the characteristics of the data set and of the algorithms acting on it.

Many methods for partitioning and mapping are described in the literature. Until recently these problems were mainly studied in the context of distributed systems. In this paper we survey some techniques for partitioning and mapping that do take into account typical properties and requirements of numerical grid-oriented problems. This is illustrated with some of our own research results.

In section 2 we introduce several types of grids that arise in solving partial differential equations. In section 3 we state the load balancing problem for grid-oriented problems, with emphasis on iterative solution algorithms, and we derive criteria that must be satisfied during partitioning and mapping. In section 4 we review some partitioning and mapping techniques. Subsequently, two techniques are discussed in more detail : recursive bisection based on a cost function (section 5) and genetic algorithms (section 6).

2. Grid-Oriented Applications

2.1. Grids and Grid Data Structures

The numerical solution of partial differential equations (PDEs) requires a discretisation using e.g. finite differences, finite volumes or finite elements. The continuous domain on which the PDE must be solved is replaced by a discrete grid of points (or cells or elements). In this paper we use *point* as a generic name for the data associated with a grid point, finite volume cell or finite element. Thus a 'point' is the basic data item. The word *grid* refers both to the discrete domain and to the data structure associated with it, i.e. the set of points.

We now describe different types of grids that often arise in PDE computations. This will allow to assess some of the requirements and the difficulties of load balancing grid-oriented problems. We discuss here only two-dimensional grids, but the extension to three dimensions is straightforward.

Regular grids

We call a grid *regular* or *structured* when (a) the boundary is rectangular or logically rectangular and (b) each point has exactly four neighbours, except at the boundaries. Thus regular grids are represented by data structures in which a point can be localised easily via indices.

A logically rectangular grid is shown in Fig. 1a. Regular grids are appropriate for many applications. Load balancing techniques for regular grids are well known [13].

Irregular grids

For a number of applications, the use of *irregular* (i.e. non-regular) grids can be required, for example

- in the case of complex domain boundaries,
- in solution-adaptive algorithms, in which the grid is refined locally during the calculation.

We can distinguish several types of irregularities.

A *block-structured grid* consists of a union of structured subgrids, called blocks, see Fig. 1b. In general, numerical methods deal with each block separately. Thus the application imposes already a partitioning of the grid and load balancing consists only of mapping the blocks onto the processors.

In an *unstructured grid* a point can have a varying number of neighbours. But also a grid that consists of triangular elements or cells as shown in Fig. 2a is unstructured.

We call a grid *internally-regular* when only condition (a) for regular grids is violated, see Fig. 2b.

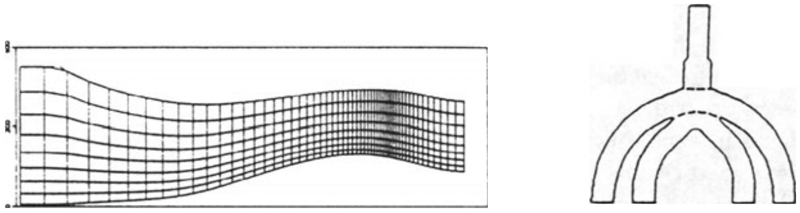


Fig. 1. a) Logically rectangular grid; b) Block-structured grid

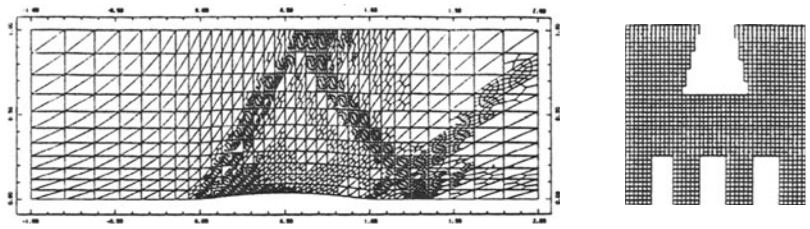


Fig. 2. a) Unstructured grid (adaptively refined); b) Internally-regular grid

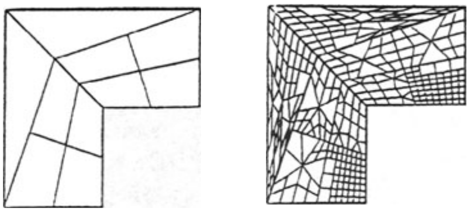


Fig. 3. Unstructured multi-level grids : two consecutive grids

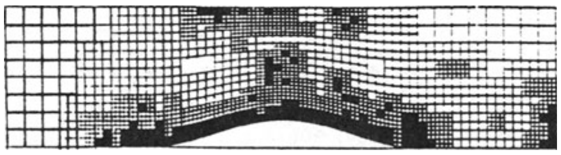


Fig. 4. Composite grid (grids of all levels are superimposed)

MULTI-LEVEL GRIDS

In multi-level or multigrid algorithms a hierarchy of grids is used.

- A hierarchy of *structured multi-level grids* is obtained when, starting from a (fine) regular grid, coarser grids are obtained by decreasing the number of grid points in each direction by a factor of two [6].
- Multi-level algorithms can also be defined on *unstructured multi-level grids*, i.e. a hierarchy of unstructured grids (see e.g. [21]). In Fig. 3 two consecutive grids of such a hierarchy are shown.
- Some multi-level techniques use *composite grids*, a hierarchy of grids in which the finer grids do not cover the complete domain but are obtained by refining a part of the next coarser grid, [6] [22], see Fig. 4. One can distinguish composite grids in which at each level a union of regular grids is used (see e.g. [25]) and composite grids in which each level consists of a union of 'internally regular' grids (see e.g. [30]).

STATIC VERSUS ADAPTIVE GRIDS

Grids can be *static*, i.e. they do not change during the computation, or can be *adaptively refined* during the calculation itself. In the latter case mostly unstructured grids are used. We will see that somewhat different load balancing approaches are necessary for static and adaptive grids.

2.2. Characteristics of Iterative PDE Solvers

In order to develop load balancing strategies for grid-oriented problems, one must not only take into account the type of the grid, but also characteristics of the solution procedure and the resulting data dependencies. We restrict our attention to iterative algorithms for the solution of PDEs, for which the following holds.

1. Each grid point represents (approximately) the same computational cost. Hence the calculation cost can be modeled accurately and a balanced calculation load can easily be achieved.
2. Iterative algorithms mainly consist of 'local operations'. The calculation associated with a grid point $e \in \mathcal{E}$ requires information from a small number of other points, called the *neighbours* $\mathcal{A}(e) \subset \mathcal{E}$. In most cases these neighbours lie close to the point e, hence this data dependency relation has a geometrical interpretation.
   It is thus natural to group grid points into *units* $u \in \mathcal{U}$, each consisting of a (preferably) connected set of grid points. Formulated alternatively, the partitioning should split the physical domain into connected subdomains.
   The data dependencies between points induce a similar relationship between units. The set of 'neighbouring units' is denoted by $\mathcal{A}(u) \subset \mathcal{U}$. When one reformulates a single-grid algorithm in terms of these units, the required communication is often restricted to the perimeter of the units or subdomains.
   In multi-level algorithms some parts of the algorithm transfer information between grids. Hence, besides intra-grid data-dependencies also inter-grid depen-

dencies occur. The latter may cause a communication volume proportional to the number of grid points. Thus for multi-level grids, units of consecutive grids, corresponding to the same part of the physical domain, should be mapped as much as possible onto the same processor [2][10].

3. The numerical efficiency of many algorithms depends on the 'grain size' of the parallelism. In grid-oriented problems, the 'grain size' can refer e.g. to the size of the subdomains on which the algorithm acts. A large grain size may lead to a lower arithmetic complexity or, in case of iterative algorithms, a faster convergence rate. As a result, large 'units for calculation' can be advantageous. For example, Red-Black Gauss-Seidel relaxation may converge substantially slower than block-wise relaxation using a lexicographic ordering of the points within each block [2][29].

3. Load Balancing Grid-Oriented Problems

3.1. The Load Balancing Problem

The '*general load balancing problem*' can be defined as the problem of scheduling all operations so as to minimise the total execution time. Scheduling implies determining when (scheduling in time) and where (assignment to a processor) each operation is executed. Considered as an optimisation problem, load balancing requires the definition of an objective function that models the execution time. Even with simple cost models the load balancing problem is NP-complete.

The difficulty can be alleviated by splitting the data parallel algorithm into a sequence of *phases.* Each phase consists of operations that can be executed concurrently, without restrictions caused by data dependencies. We refer to [9] for a more precise definition of phase. The general load balancing problem can then be replaced by a series of *'restricted load balancing problems'*, one for each phase. It is obvious that the total execution time obtained in this way cannot be smaller than by solving the general load balancing problem, since we restrict the space of feasible solutions. On the other hand, load balancing phase per phase reduces the size of the optimisation problem. This makes it easier to find a (sufficiently) good solution.

The concepts 'phase' and 'load balancing phase per phase' are well suited for algorithms for solving PDEs. For example, in the Red-Black Gauss-Seidel relaxation method, each update of all Red (resp. Black) points can be considered as a phase. Different phases may require a different partitioning and mapping. When a grid has been adaptively refined, the workload and the data dependencies have been changed and the calculation of a new partitioning and mapping can be advantageous. In multi-level algorithms the data dependencies (and thus the communication patterns) during the relaxation phase ('intra-grid communication') are completely different from those during the intergrid transfer phases (restriction and interpolation), resulting in 'inter-grid communication'.

The remaining optimisation problem is still not trivial. There is a conflict between the minimisation of the calculation cost (or calculation load imbalance) and the minimisation of the communication cost. When the ratio between calculation cost and communication cost — which is application and machine dependent — is high, it can be sufficient to minimise only the calculation cost; otherwise the complete minimisation problem must be considered.

Modeling the communication time is a major difficulty. The total communication time depends on the *communication volume* but also on the *number of messages* because of the 'message startup time', on the *communication distance* (the number of physical links on the communication path) and on the *link contention.* Especially the latter aspects are hard to model.

3.2. Load Balancing Grid-oriented Iterative Algorithms

The following characteristics of grid-oriented iterative algorithms should form a basis for load balancing techniques :

- The algorithm can be split in phases ; the execution of each phase is initiated by the application program at well-defined moments. Often the calculation cost associated with a phase is high.
- The workload and communication requirements for each unit do not change during a phase and can be predicted rather accurately

Load balancing can thus be achieved by an appropriate partitioning and mapping of the data for each phase.

When the algorithm consists of only one phase and when a static grid is used, partitioning and mapping must be done only at the beginning of the calculation ('static load balancing'). This pre-processing step may be expensive. When a data parallel program consists of several phases, e.g. when multi-level algorithms or adaptive refinement procedures are used, a different data distribution may be required for each phase. When the execution time for each phase is relatively large, it may be advantageous to re-distribute the data and the associated work by a static load balancing procedure at the beginning of each phase. This is only feasible when the execution time of the load balancing algorithm is low.

In case of adaptive refinement, re-balancing the load after each refinement is called 'iterative static load balancing' [11] (or 'quasi-dynamic load balancing' [31]). Ideally the new data distribution should not differ too much from the previous one, in order to keep the remapping cost low.

Note that under these conditions truly 'dynamic load balancing' [1][3] is not necessary.

Each phase of a grid-oriented application can be load balanced either by

A) *allocation* of grid points to *units* (= *partitioning*) and subsequent *mapping* of the units onto the processors,
 or

B) *mapping* the individual grid points onto the processors; in this case the set of units is equal to the set of points.

In both cases the units are considered to be atomic for all operations (calculation and communication) within the current phase.

During the partitioning step, i.e. the definition of units, various types of restrictions can be taken into account — besides those aiming at load balancing :

- for 'block algorithms' (with good numerical properties) a unit should be an interconnected set of points,
- representation of units by regular data structures can be mandatory, e.g. to allow vectorisation,
- in multi-level algorithms, the definition of units on the finer grids can impose conditions for the definition of units on coarser grids (or vice versa) [10].

In these situations partitioning often cannot be automated and should be done by the application (programmer).

We now formulate some criteria that must be satisfied during partitioning and mapping. These criteria must be translated mathematically into cost functions to be minimised or they must form the basis for load balancing heuristics. We distinguish two different situations, depending on the number of units compared with the number of processors.

1) THE NUMBER OF UNITS IS EQUAL TO THE NUMBER OF PROCESSORS

In this case, the following criteria must be satisfied approximately [7] :

- during partitioning of the grid :
  1. The partitions or units have approximately the *same number of points.* Assuming that the workload is equal for all points, this ensures load balance.
  2. Each partition consists of only *one interconnected set of points.* Because most algorithms perform local operations, neighbouring points should be allocated to the same processor as much as possible in order to minimise communication. For some algorithms, this also leads to higher numerical efficiencies.
  3. The *perimeter* (i.e. the number of points at the boundary between units) is *small.* In general this ensures a low communication volume, since for most problems the communication consists of exchanging data corresponding to the boundaries of the subdomains.
  4. The *number of neighbouring units is minimum.* Typically, the number of messages is proportional to the number of adjacent subdomains.
- during mapping of units to processing elements :
  1. The communication cost between the units is minimal on the given parallel machine. This requires that the architecture of the parallel machine (e.g. hypercube, mesh, ...) is taken into account by considering the communication distance.
  2. The synchronisation among units is low.

Note that this distinction between the partitioning and the mapping steps is somewhat artificial and often both groups of criteria are treated together, as in strategy B) (see above).

In case of iterative static load balancing either a completely new partitioning must be calculated or the previous partitioning must be adapted by '*cut-and-paste*' operations. The cut-operation splits off a part of a unit with a high workload; the paste-primitive adds this part to a unit in a less loaded processor, see [11].

2) THERE ARE MUCH MORE UNITS THAN PROCESSORS

This situation typically arises when the units are defined by the application. No further partitioning step has to be done when static grids are used. In case the grid is adaptively refined, the workload associated with some units may become much larger than the load associated with other units. Then a '*split*' operation, which splits a unit into several new units, can be executed [11]. This enables better solutions for the mapping problem.
The units must be mapped onto the processors such that each processor receives approximately the same workload, and such that the criteria for partitioning and mapping, given above, are satisfied as much as possible.

4. Partitioning and Mapping Techniques

In this section we give an overview of some important partitioning and mapping techniques for grid-oriented problems.

The set of units (or points) and the data dependencies can be described by a *Unit Interaction Graph* $(\mathcal{U}, \rightarrow)$ [11], also called *Task Interaction Graph* [28][27]. The nodes of this graph represent the units $u \in \mathcal{U}$. The neighbouring relationship between the units is represented by the graph edges. *Partitioning* the grid is equivalent to the graph partitioning problem for the UIG. The mapping problem can also be formulated using a graph-theoretical formulation. Note that when the processors of the parallel computer are fully interconnected, the mapping problem is identical to grid (or graph) partitioning.

When we denote the set of processors by $\mathcal{Q}$ and the set of units by $\mathcal{U}$, the solution to the mapping problem can be represented as a function ('mapping') $m : \mathcal{U} \rightarrow \mathcal{Q}$, that assigns unit u to processor $m(u)$.

Partitioning and mapping techniques can differ in various ways.

- The solution strategy : both the solution of the partitioning problem and the computation of the mapping m can be obtained by explicit minimisation of a cost function or by an heuristic approach.
- Global or local mapping : using information about all units theoretically leads to the best result. Local mapping methods use only information about units in a processor subset; this approach results in easier to solve problems and is certainly cheaper.

– Centralised or distributed implementation : both local and global mapping techniques can benefit from a parallel implementation of the algorithm.

In the graph $(\mathcal{U}, \rightarrow)$ weights can be assigned to graph nodes and edges, giving information about the cost involved in the corresponding calculation and communication. This information is sufficient to construct simple cost models. Let P be the number of processors and N the number of units. Let t_{calc} be the time to perform one floating point operation on the particular machine and let c_i be the number of floating point operations to be performed for unit u_i. Then the calculation cost associated with u_i is $\Lambda_{ii} = t_{calc}c_i$. Let t_{comm} be the time to send one word between neighbouring processors and let c_{ij} be the number of words to be sent from u_i to u_j. Then the communication cost can be approximated by $\Lambda_{ij} = t_{comm}c_{ij}$. The set of units assigned to processor q is denoted by $\mathcal{U}^{(q)}$. $\Delta(p,q)$ represents the graph distance between processors p and q, i.e. the minimal number of graph edges that have to be traversed in order to go from p to q. The distance between the processors holding units u_i and u_j is denoted by Δ_{ij}.

In section 5 we will use the following simple cost functions to solve the mapping problem :

$$C_1(m) = \max_{q\in\mathcal{Q}} \sum_{u_i\in\mathcal{U}^{(q)}} \Lambda_{ii}$$

$$C_2(m) = \max_{q\in\mathcal{Q}} \left(\sum_{u_i\in\mathcal{U}^{(q)}} \Lambda_{ii} + \max_{\substack{u_i\in\mathcal{U}^{(q)}\\ u_j\in\mathcal{A}(u_i)}} \Lambda_{ji}\Delta_{ji} \right)$$

Cost function C_1 can be considered to be the model of a fully connected machine with infinite communication bandwidth and zero startup time. C_2 assumes a finite communication speed and takes into account the machine topology via the Δ_{ji} factor. This will force the maximum distance between the processors onto which neighbouring units are mapped to be minimal, at least if this does not endanger the calculation load balance.

We now present a classification that is based on the solution strategy.

A) OPTIMISATION TECHNIQUES BASED ON A COST FUNCTION

1. *Branch-and-bound algorithms :*
   The assignment of points or units to the processors is a discrete optimisation problem that can be solved by a branch-and-bound algorithm. The solution space has cardinality N^P. Because of the large search space, enumeration of all possible solutions is prohibitively expensive. However, in general some symmetry properties that are present in the cost function and the underlying hardware can be used to reduce the search space, see e.g. [9]. In spite of these simplifications, branch-and-bound algorithms — while being exact — remain too expensive in practice.

2. *Simulated annealing techniques :*
Simulated annealing is a very general optimisation method which stochastically simulates the slow cooling of a physical system. The idea is that there is a cost function C which associates a cost with a state of the system, a 'temperature' T and various ways to change the state of the system, see e.g. [31] [14] [15]. Changes are proposed and the resulting changes ΔC in C are evaluated in an iterative procedure, while the temperature T gradually decreases to zero. A change is accepted or rejected according to the Metropolis criterion : if the cost function decreases the change is accepted unconditionally, otherwise it is accepted with probability $\exp(-\Delta C/T)$. Under certain conditions ('reachability', sufficiently slow decrease of the temperature) the probability that the global optimum is found tends to certainty.
R. Williams [31] did extensive experiments with simulated annealing for solving the partitioning problem. He observed that this approach is rather sensitive to the choice of a number of parameters of the method (the set of possible changes, the cooling rate, ...). For sufficiently slow cooling this method produces good solutions, but then the method is very expensive. Note also that effective parallelisation of simulated annealing is not trivial. We refer to [31] for more information.
3. *Evolution algorithms*
Another approach to the graph partitioning problem is proposed by Mühlenbein et.al. [23]. The mapping problem is transformed into a continuous optimisation problem by representing the allocation by a matrix x, in which $x_{ij} \in [0,1]$ denotes the probability that unit u_i is allocated to processor q_j. The cost function
$$C(m) = \left(\prod_{q_k \in \mathcal{Q}} \sum_{u_i, u_j \in \mathcal{U}^{(q)}} \Lambda_{ij} x_{ik} x_{jk} \right)^{-1}$$
represents the intra-partition connectivity, which should be maximised. This cost function ignores the machine topology and is therefore better suited for decomposition than for mapping. For this (continuous) cost function, the steepest descent algorithm turns out to be very efficient.
4. *Genetic algorithms*
Genetic algorithms resemble simulated annealing in that they are also very general and robust optimisation methods that simulate an optimisation process found in nature. More specifically, genetic algorithms simulate the processes of reproduction, crossover and selection that make living beings optimally adapted to their environment. Genetic algorithms in the context of load balancing are further discussed in section 6.
5. *Mapping based on minimising a cost function after projection*
Chrisochoides et.al. [7] developed a mapping method in which the nodes of both the UIG and the processor graph are first projected onto an Euclidean space based on eigenvectors of a matrix related to the adjacency matrix. The mapping is then determined by minimising a cost function defined in this

Euclidean space (related to the distance between the projected nodes of both graphs).

B) Clustering techniques

Rather than trying to minimise both computational workload imbalance and communication simultaneously, the search space can be reduced if only one of both terms is explicitly modeled while the other is used implicitly in guiding the search [27].

Some authors have proposed partitioning and mapping strategies based on '*clustering techniques*'. In these approaches clusters of units are formed with high intra-cluster dependencies and low inter-cluster communication. These clusters form a partitioning of the data set and can be considered as units for subsequent mapping. The clustering is based on some sorting of the grid points and subsequent partitioning.

Sadayappan et. al. [26] [27] propose a 'nearest-neighbour mapping' algorithm, that proceeds in two steps :

1. An initial mapping is generated by grouping nodes of the UIG in clusters and assigning clusters to processors so that the nearest-neighbour property is satisfied, i.e. neighbouring nodes are assigned either to the same processor or to neighbouring processors.
2. The initial mapping is successively modified using a boundary refinement procedure where nodes are reassigned among processors in a manner that improves calculation load balance but always maintains the nearest-neighbour property.

Thus the nearest-neighbour mapping scheme explicitly attempts to minimise calculation cost, while low communication costs are achieved implicitly by the search strategy.

Farhat [12] proposes to sort the nodes of the UIG based on the Reverse Cuthill-McKee ordering scheme. This scheme is well known in sparse matrix techniques, where it is used to reduce the bandwidth and the profile of a matrix by re-ordering the equations and the unknowns of a linear system.

C) Recursive bisection heuristics

A heuristic frequently applied to the graph partitioning problem is recursive bisection. In this approach P-way partitioning is replaced by repeated 2-way partitioning (bisection). Such a procedure is defined by a recursion with depth $\log_2 P$.

Although recursive bisection is used most often for partitioning [15] [13] [28] [27] [31]. it can also be applied to the mapping problem [7] [9]. In this case both the UIG and the processor graph are split recursively; the mapping assigns a partition of the UIG to the corresponding processor. This technique is applicable to a fully connected parallel computer, to a mesh of processors, and to hypercubes. In the latter case there exists a fixed correspondence between the partitions as generated by the recursive procedure and the recursive numbering of the processors, see [9].

Another approach for mapping partitions generated by a recursive bisection method is proposed by Sadayappan et.al. [27] and is based on the Kernighan-Lin mincut heuristic [20]. Starting with an initial mapping, pairs of partitions are swapped in sequence, attempting to minimise the communication cost (taking into account distances between processors).

Particular recursive bisection heuristics differ in the method used to split a partition. We now describe some of the possible splitting techniques.

We first assume that the weights of all nodes and edges of the UIG are *identical.* This is for example the case when each unit represents one point of the grid. We can associate a scalar quantity $s(u)$ with each node u of the UIG. Following Williams [31], we call $s(u)$ a separator field. By evaluating the median S of the separator field, we can bisect the graph, according to whether $s(u)$ is greater or less than S. This ensures that the calculation cost (i.e. the number of nodes) in each half is equal. The separator field is chosen so that the communication is minimised. Some of the possibilities are described below.

– *geometry based splitting*
  A simple and cheap choice for the separator field $s(u)$ is based on the geometrical position of the corresponding grid point (or unit). We might let the value of $s(u)$ be the x-coordinate associated with the grid point (the x-coordinate of the center of an element in case of finite elements). Then the grid is split in two by a median line parallel to the y-axis. If this procedure is repeated recursively, we obtain a partitioning of the grid into 'strips'.
  In the Orthogonal Recursive Bisection method (ORB), $s(u)$ is chosen to be alternatingly the x- and the y-coordinate of the point [13] [31]. This technique implicitly ensures a low inter-processor communication volume. Indeed, for regular meshes ORB will lead to (almost) square partitions, which minimises the perimeter of the partitions, ensuring minimal inter-partition communication cost in case of local algorithms.
  Experiments of R. Williams [31] show that for irregular meshes ORB can lead to rather high communication costs, because it pays no attention to the connectivity of the graph. Note that the partitions obtained by ORB do not always consist of connected sets of grid points. This is illustrated in Fig. 5
  ORB is incorporated into the software package DIME ('Distributed Irregular Mesh Environment') of R. Williams [32], that is designed for applications on an irregular grid of triangular cells. The package deals with all communication aspects, allows adaptive refinement and performs iterative static load balancing.

– *splitting based on eigenvectors of the Laplacian matrix of the graph*
  A better method for splitting a graph is based on the eigenvector corresponding to the second lowest eigenvalue of the Laplacian matrix of the graph [31] [4] [8] [24]. This Laplacian matrix is derived from the adjacency matrix of the graph. In general, this technique yields lower communication costs than

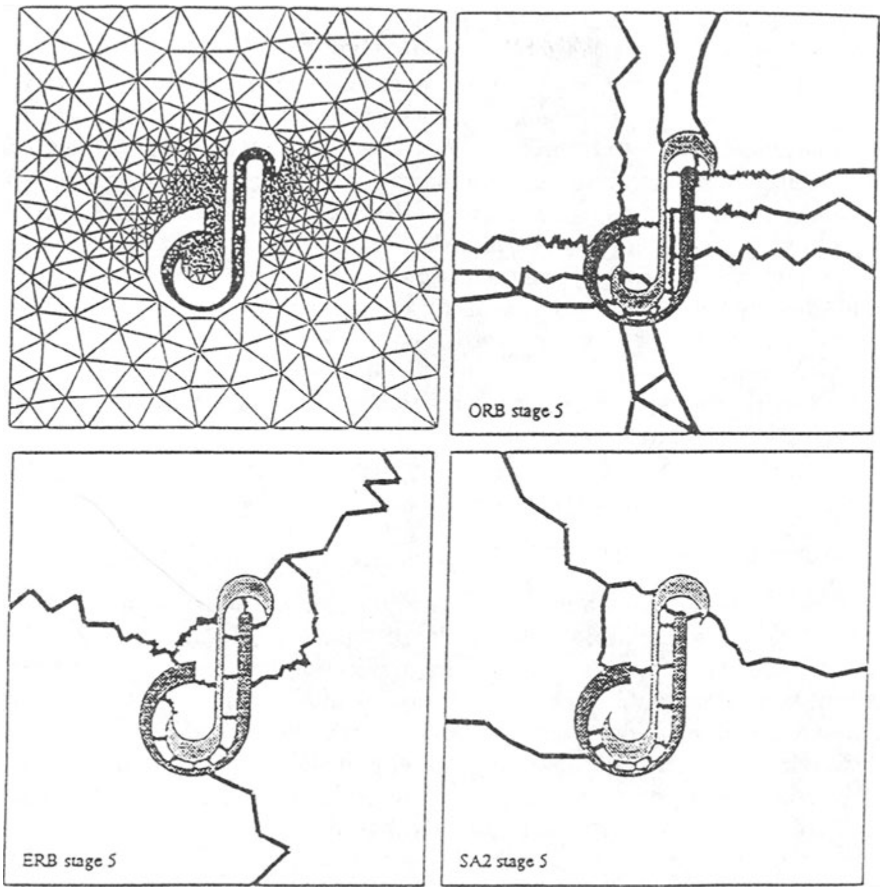


Fig. 5. Unstructured triangular mesh, heavily refined between the two 'holes'. Partitioning in 16 units by Orthogonal Recursive Bisection (ORB), Eigenvector Recursive Bisection (ERB) and a simulated annealing approach (SA2). This figure is reprinted from [31].

ORB (see also Fig. 5). An important advantage is that partitioning in connected sets of points is guaranteed. However, Eigenvector Recursive Bisection is more expensive that ORB (a factor of 2 is reported in [31]) and the number of elements migrated can be much larger than with ORB, if these bisection algorithms are used to rebalance the load after refinement [31].

The next two splitting techniques allow that *different* weights are associated with the nodes and the edges of the UIG. Thus these techniques can be used to solve the mapping problem when the data set is already partitioned in units.

- *splitting based on a clustering heuristic*
  Sadayappan et.al. [27] propose to create an initial load balanced two-way partitioning. This can be done e.g. by assigning the nodes of the UIG, one by one, always to the partitioning with lesser total calculation load. Afterwards an heuristic is used to transfer nodes between the two partitions of the UIG to minimise the inter-partition communication volume, while maintaining load balance. The heuristic is based on the Kernighan-Lin mincut algorithm [20]. Note that if all edges of the UIG represent the same communication volume, minimal inter-partition communication is achieved by a 'minimal cut' of the inter-partition edges. This approach can be expensive.
- *splitting based on a cost function*
  The bisection may be based on the minimisation of a cost function, but now applied recursively to the 2-way partitioning case. This approach will be discussed in the next section.

5. Recursive Bisection based on a Cost Function

5.1. Bisection Heuristic

In this section we describe a recursive bisection heuristic based on a cost function in more detail. We also present the results of some experiments.

We consider the situation in which the application has already partitioned the problem data set in units, each of which may require a different amount of calculation and communication. Each bisection step can then be based on minimising a cost function for the 2-way partitioning case. Even if each 2-way problem is solved exactly, recursive bisection does not yield the global optimum. It is thus not necessary to solve the 2-way partitioning exactly, but one can use a cost based heuristic like the following one. A set of units $\mathcal{U}$ can be bisected by sequentially allocating a unit u to one of both partitions such that the cost associated with the partial allocation is minimal. The assignment of a unit to a machine partition is never undone; no backtracking is performed.

An efficient algorithm is based on the following observation. Suppose that $\mathcal{V} = \bigcup_{q\in\mathcal{Q}} \mathcal{V}^{(q)}$ is a set of t units which are already allocated. For both cost functions $C_k (k = 1, 2)$ it is possible to define a *partial cost function* PC_k with the properties

$$PC_k(m,\mathcal{U}) = C_k(m)$$
$$\mathcal{V}_a \subset \mathcal{V}_b \Rightarrow PC_k(m,\mathcal{V}_a) \leq PC_k(m,\mathcal{V}_b)$$

An example is :

$$PC_2(m,\mathcal{V}) = \max_{q\in\mathcal{Q}} \left(\sum_{u_i\in\mathcal{V}^{(q)}} \Lambda_{ii} + \max_{\substack{u_i \in \mathcal{V}^{(q)} \\ u_j \in \mathcal{A}(u_i)\cap\mathcal{V}^{(q)}}} \Lambda_{ji}\Delta_{ji} \right)$$

Let u be a unit that is not yet allocated. It is now possible to evaluate the costs associated with both possible allocations of u in an incremental way. For example, PC_2 can be written for bi-partitioning as :

$$PC_2 = \max_{k=1,2}(C_{calc}^{(k)} + C_{comm}^{(k)})$$

with

$$C_{calc}^{(k)} = \sum_{u_i \in \mathcal{V}^{(q)}} \Lambda_{ii} \qquad C_{comm}^{(k)} = \max_{\substack{u_i \in \mathcal{V}^{(q)} \\ u_j \in \mathcal{A}(u_i) \cap \mathcal{V}^{(q)}}} \Lambda_{ji}\Delta_{ji}$$

Given the costs associated with partial allocation $\mathcal{V}_{i-1}$, the costs for $\mathcal{V}_i = \mathcal{V}_{i-1} \cup \{u_i\}$ when u_i is allocated to partition $l \in \{1,2\}$ can be calculated as :

```
if $k = l$ then
    $C_{calc}^{(k)}(\mathcal{V}_i) = C_{calc}^{(k)}(\mathcal{V}_{i-1}) + \Lambda_{ii}$
    $C_{comm}^{(k)}(\mathcal{V}_i) = \max\{C_{comm}^{(k)}(\mathcal{V}_{i-1}), \max_{u_j \in \mathcal{A}(u_i) \cap (\mathcal{V}_{i-1} \setminus \mathcal{V}_{i-1}^{(k)})} \Lambda_{ji}\}$
else
    $C_{calc}^{(k)}(\mathcal{V}_i) = C_{calc}^{(k)}(\mathcal{V}_{i-1})$
    $C_{comm}^{(k)}(\mathcal{V}_i) = \max\{C_{comm}^{(k)}(\mathcal{V}_{i-1}), \max_{u_j \in \mathcal{V}_{i-1}^{(k)}, u_i \in \mathcal{A}(u_j)} \Lambda_{ij}\}$
end if
```

Such heuristics yield solutions of which the practical quality is limited by the accuracy of the cost function and the minimisation technique. The solution was ameliorated by applying an iterative improvement technique, consisting of a number of steps of the Metropolis algorithm [7] [9].

5.2. Numerical Experiments

A software tool LOCO [9] [11] has been developed that assists the execution of data parallel programs. LOCO allows any problem data structure, but the application has to define the units and the UIG for each phase. It schedules all calculations, communication and load balancing. Several mapping algorithms have been incorporated into this tool. In order to compare them, a model problem was solved using LOCO.

Consider the parabolic partial differential equation defined for $x \in [0,1]$ and $t \in [0,\infty)$:

$$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2} \qquad u(0,t) = 0; u(1,t) = 0; u(x,0) = 1$$

This equation can be discretised in time using an explicit linear one-step method (forward Euler), and in space with finite differences. Let there be n equidistant points in the interior of $[0,1]$, separated by a distance $h = \frac{1}{n+1}$. If the time step is chosen as $k = \frac{h^2}{3\alpha} < \frac{h^2}{2\alpha}$ (stability condition) the explicit method is :

$$u_i^{k+1} = \frac{1}{3}(u_i^k + u_{i+1}^k + u_{i-1}^k)$$

which requires 3 floating point operations for each interior point at each time level.

A data parallel algorithm is obtained by partitioning the set of n discretisation points. Suppose there are P processors and $B \cdot P$ units. In order to experiment with different load balances, the number of consecutive interior points n_i assigned to each unit is determined by

$$\begin{aligned} n_i &:= \lfloor \frac{n}{BP}(1 + \frac{2\sigma}{BP}(i - \frac{BP}{2})) \rfloor \qquad i > 0 \\ n_0 &:= n - \sum_{i>0} n_i \end{aligned}$$

in which $\sigma \in (-1,+1)$ determines the variation in block sizes. Note that for this special way of partitioning an almost perfect load balance is possible when B is even, by assigning blocks i and $BP - i$ to the same processor. Each time step corresponds to a phase. The calculation that must be applied to each unit consists of updating the values in all its points. The data exchange consists of sending the value in the points on either side of each interval. The unit interconnection graph has the topology of a chain.

While for a one-dimensional problem partitioning is easier than for the two-dimensional case, the mapping problem has the same complexity. The amount of computation per discretisation point is however relatively small.

Let $t_j{}^{phase}$ denote the time needed by processor q_j to complete a phase, and $t_i{}^{calc}$ the calculation time for unit u_i. The total time spent in the phase is :

$$T(P,n) = \sum_{q_j \in \mathcal{Q}} t_j{}^{phase}$$

The calculation load balance is defined by :

$$\lambda(P,n) = \frac{\sum_{u_i \in \mathcal{U}} t_i{}^{calc}}{P \cdot \max_{q_j \in \mathcal{Q}} \sum_{u_i \in \mathcal{U}_j} t_i{}^{calc}}$$

The *effectivity* is the fraction of time spent in the actual calculation :

$$\alpha(P,n) = \frac{\sum_{u_i \in \mathcal{U}} t_i{}^{calc}}{T(P,n)}$$

It is evident that $\alpha \leq \lambda$. The difference is due to scheduling and communication overhead The parallel efficiency

$$\epsilon_p(P,n) = \frac{T(1,n)}{P \cdot T(P,n)} = \frac{\alpha(P,n)}{\alpha(1,n)}$$

and the scaled parallel efficiency

$$\epsilon_s(P,n) = \frac{T(1,n)}{T(P,P\cdot n)} = \frac{\alpha(P,P\cdot n)}{\alpha(1,n)}$$

indicate the amount of parallelism in the program. They are related by $0 \leq \epsilon_p(P,n) \leq \epsilon_s(P,n) \leq 1$.

5.3. Comparison of Mapping Algorithms

The test problem was run on an iPSC/2 hypercube using different mapping algorithms for global optimisation. We used $n = 30000$ interior points and $P = 8$ processors, and a variable number of blocks per processor $B = 1, 2, 4$. In each data exchange 1 double precision number must be sent ($\Lambda_{ij} \approx 350\mu s$). For each grid point the calculation requires 3 floating point operations ($\Lambda_{ii} \approx \frac{3n}{BP} 17\mu s$).

First the perfectly balanced case was studied with exact mapping (cost function C_2, branch-and-bound) in order to estimate the overhead encurred by using LOCO and the delay due to communication. This was compared with a non-balanced partitioning ($\sigma = 0.4$).

	$B = 1$	$B = 2$	$B = 4$
$\alpha(\%)(\sigma = 0)$	85	81	74
$\alpha(\%)(\sigma = 0.4)$	72	81	74

Both the overhead and the communication delay increase as the number of blocks is augmented, and constitute a significant percentage of the total time. Because of the special kind of partitioning, the optimal assignment is only imbalanced in case $B = 1$.

We now consider the total cost, load balance and effectivity that were obtained with recursive bisection for cost functions C_1 and C_2 (RB_1 and RB_2), the evolution algorithm [23] ($EVOL$) and their counterparts with iterative improvement (RBI_1, RBI_2, $EVOLI$).

$\sigma = 0.4$	$B = 1$				$B = 2$				$B = 4$			
	C_1	C_2	λ	α	C_1	C_2	λ	α	C_1	C_2	λ	α
RB_1	122	123	77	67	101	102	93	75	95	98	99	74
RBI_1	122	123	77	67	96	98	98	81	94	97	99	72
RB_2	122	123	77	67	101	102	93	75	95	98	99	73
RBI_2	122	123	77	68	96	98	98	81	94	97	99	74
$EVOL$	122	123	77	71	136	138	69	60	124	127	76	60
$EVOLI$	141	142	77	60	103	106	91	76	97	99	96	70

For the case $B = 1$ all methods succeed in finding a 1-to-1 assignment of units to processors. These assignments usually have a worse effectivity than for the optimal assignment ($\alpha = 72\%$) because the mapping algorithms fail to find the optimal allocation with respect to communication. For $B = 2$ and $B = 4$ all methods yield a reasonable assignment with an effectivity close to the optimal one, except $EVOL$ and $EVOLI$. Due to their stochastic nature the latter sometimes find a good assignment, and sometimes don't. Iterative improvement always is worth while, although it is time consuming for large problems. Iterative improvement of a solution obtained by $EVOL$ takes longer because this starting solution tends to be worse.

For a problem of fixed size $n = 30000$, partitioned unevenly ($\sigma = 0.4$) in twice as much blocks as there are processors ($B = 2$), the parallel efficiency was measured with $P = 1, 2, 4, 8$ and 16 processors. Fig. 6 shows how the efficiency degrades as P increases. This is due to the increasing importance of communication, difficulty of mapping, and overhead.

In Fig. 7 the scaled efficiency is shown for the case of fixed average block size $n/BP = 15000$. In this case the relative importance of calculation and computation is constant, thus giving a more realistic impression of the scalability of the algorithm. The deterioration of ϵ_s with increasing machine size is much less severe.

5.4. Local Mapping

The time invested in the calculation of a new allocation can be limited by resorting to non-global optimisation. In this case load is redistributed only within certain processor subsets.

Some experiments for the hypercube case are summarised in the following table. Let l be the hypercube dimension. The scaled efficiency was determined on the iPSC/2 for the case of two blocks per processor ($B = 2$) with $n/BP = 15000$. The load balancing strategy consisted of two steps. In a first step blocks of unequal size were generated ($\sigma = 0.4$) and distributed among the processors assuming that their calculation load is equal (communication costs were ignored) using RBI_1. In a second step local remapping was done with RBI_2 for g-dimensional subcubes ($g < l$).

RBI_2		P				
$\epsilon_s(\%)$		1	2	4	8	16
	0	100	99.2	94.0	90.0	88.0
	1		99.1	99.2	97.5	95.5
g	2			99.0	96.7	96.7
	3				96.4	96.5
	4					95.9

Although the quality of the achievable optimal mapping increases with g, the optimisation problem becomes more difficult, and the heuristic may perform worse. Therefore global optimisation is not always the best.

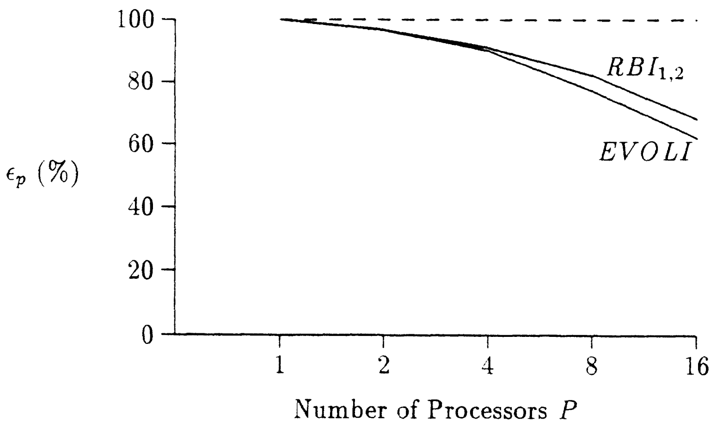


Fig. 6. Parallel efficiency ϵ_p

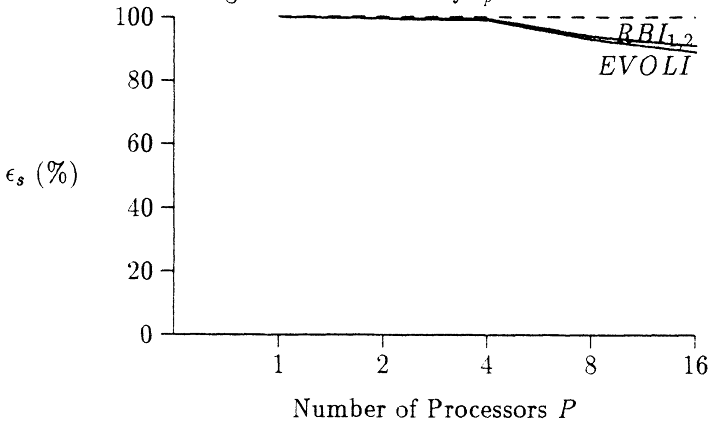


Fig. 7. Scaled efficiency ϵ_s

6. Partitioning and Mapping with Genetic Algorithms

6.1. Introduction

Genetic algorithms are robust optimisation methods, based on an analogy with the way individuals in nature adapt to their environment. Since their introduction by John Holland [19], they have been used successfully in a wide range of applications (an extensive overview can be found in [16]).

In these algorithms, one lets evolve a population of individuals represented by chromosomes. In every generation, the fitness of the individuals is evaluated according to a given criterion. To produce the next generation, parents are selected in the population to mate and produce offspring. The probability for an individual to be selected for mating is proportional to its fitness. In this way, the properties of above average individuals are rapidly distributed among the members of the population while the properties of the inferior individuals vanish in the population.

In genetic algorithms, the chromosomes are normally bit strings. In order to use a genetic algorithm (GA) for the solution of an optimisation problem, on must be able to code all the possible solutions of the problem as bit strings. In most cases, fixed length bit strings are the most obvious choice but for some problems the use of variable length bit strings can be advantageous.

The children are produced by combining the chromosomes of their parents, using a number of operators inspired by phenomena found in nature. The most frequently used operators are crossover and mutation. The *crossover* operator takes two chromosomes and creates a new chromosome by taking the bits of the first chromosome at some locations and the bits of the second chromosome at the remaining locations. The *mutation* operator takes one chromosome and changes each bit with a given probability P_M.

If we represent the population at generation t ($t = 0, 1, 2, \ldots$) by $P(t)$, we can formulate the basic GA as follows :

```
create the initial population $P(t = 0)$ with $N$ individuals ;
while not stop do
    evaluate the fitness of the individuals of the population ;
    create the individuals of the population $P(t + 1)$ :
    repeat
        select two parents in $P(t)$ with the selection probability of an
          individual being proportional to its fitness ;
        produce (the chromosomes of) the children, either by applying
          crossover between the chromosomes of the parents with pro-
          bability $P_C$ or by copying them with probability $1 - P_C$ ;
        apply mutation to the chromosomes of the children with pro-
          bability $P_M$ ;
        add the children to $P(t + 1)$.
    until population $P(t + 1)$ is complete ;
    set $t = t + 1$
end while
```

6.2. Application of the GA to the Partitioning and Mapping Problem

In order to use a GA, the user must supply a suitable coding for his problem and a function that attributes a measure of fitness to a given solution. When the GA is applied to the partitioning and mapping problem, a given cost function must be minimised, so the lower the value of the cost function for an individual, the higher its fitness.

Here, we describe experiments based on the cost function, elaborated by R. Williams in [31] :

$$H = \frac{P^2}{N^2} \sum_{q \in \mathcal{Q}} N_q^2 + \mu \left(\frac{P}{N}\right)^{\frac{d-1}{d}} \sum_{u_i \leftrightarrow u_j} 1 - \Delta_{ij}$$

where N_q in the first summation is defined as

$$N_q = \sum_{u_i \in \mathcal{U}} \Delta(q, m(u_i)) = \# \mathcal{U}^{(q)}, \quad q \in \mathcal{Q}$$

and where the second summation must be performed over all couples of interacting units.

The constant μ determines the relative importance of the costs of calculation and communication : the greater μ, the more important the communication as compared with the calculations.

The constant d is the dimension of the grid from which the unit interaction graph was derived. This function is relatively simple while at the same time it captures the relevant aspects of the problem. Moreover, the form of the function is such that the GA converges relatively fast.

In Fig. 8, the results for a 'typical' run of the genetic algorithm are shown. The problem consisted in the mapping of 64 units, interconnected in an 8 by 8 regular grid, on 4 processors. The factor μ was chosen to be equal to 0.5. For this problem and for $\mu = 0.5$, the value of the cost function for the optimal mapping is equal to 8.0.

A population of 64 individuals was used, the crossover rate P_C was taken to be 1.0 and the mutation probability P_M was set to 0.01. The genetic algorithm was performed 10 times. The results shown are these of the run that produced the median of the end results. In the figure the cost function of the worst individual of the population, the mean cost function among all the individuals of the population and the cost function for the best individual are plotted against the generation number. It can be seen that after 95 generations, the whole population has converged to a solution. The solution, however, is not optimal. This *premature convergence* to a local minimum is an important problem and the improvement of the GA to avoid this is currently an important research topic (see e.g. [5]).

6.3. Improvement of the GA by Using Heuristics

It is possible to speed up the convergence substantially by heuristically improving the quality of the individuals after their creation and before they produce offspring (see e.g. [18]).

For the same problem as mentioned above, the GA without improvement of the individuals is compared with the GA with improvement. Two heuristics were used :

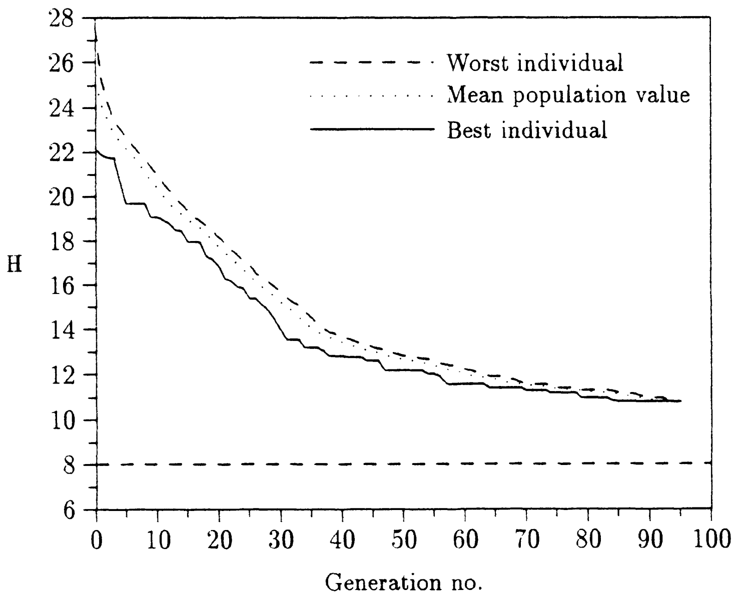


Fig. 8. Minimal, maximal and mean value of the cost function

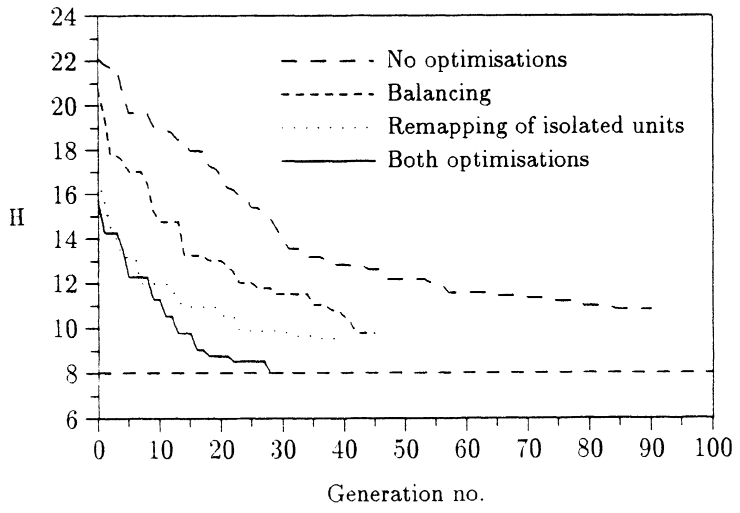


Fig. 9. Comparison of the GA with and without heuristics

1. a heuristic that balances the number of units among the processors. This heuristic assumes that the number of processors is a power of 2, say $P = 2^p$. The processor numbers are then bit strings of length p. The heuristic is then as follows :

   ```
   for $i$ is 0 to $p-1$ do
       for each pair of processors whose numbers differ in the $i$th bit do
           shift from the processor with the greater number of units half
               of the surplus of units to the other processor
       end for
   end for
   ```

   When the heuristic is performed, the units are balanced among the processors but neighbouring units will in general be mapped on different processors.
2. a heuristic that remaps isolated units, i.e. units for which the connected units are all mapped on another processor.

In Fig. 9, results are given when no heuristic is used, when only one of the heuristics is used and when both heuristics are used. In each case, the value of the cost function for the best individual in the population is plotted against the generation number. It can be seen that the algorithm converges faster (i.e. needs fewer generations) and finds better solutions when the heuristics are used. Of course when the heuristics are used, calculating a new generation involves more work but nevertheless the execution time of the algorithm is in most cases shorter. This is illustrated in the following table where the calculation times on a DECstation 3100 are reported.

Heuristics used in the GA	Execution time
No heuristics	49.297 sec
Balancing	49.843 sec
Remapping of isolated units	36.072 sec
Both heuristics	45.091 sec

6.4. Parallelisation of the GA

When a genetic algorithm is used to balance the load of a parallel computer, it is essential that it is parallelised because otherwise the load balancer becomes a bottleneck when a large number of processors is used. Besides, as the genetic algorithm is rather time consuming, even for other applications, the use of a parallel genetic algorithm can be advantageous.

The GA is inherently parallel in that the individuals of the population coexist in parallel and that couples of individuals concurrently mate and produce children, so it is logical to distribute the individuals of a population among the processors.

In the traditional GA, a panmictic population [17] is used, i.e. during reproduction each individual may choose whichever individual from the population as a mate. This means that each individual must 'know' all the other individuals of the population. More specifically, each individual must know the fitness of the other individuals to be able to make a choice among them and consequently must know the chromosome of the individual it selected as mate. This need for global knowledge inhibits efficient parallelisation. Therefore, a number of modifications have been proposed which all have in common that the number of potential mates of an individual is restricted.

In the *island models* a number of isolated populations are kept at the same time. Each population is panmictic, so that individuals can only mate within their population. The isolated populations may communicate by migration, viz. from time to time individuals move from one population to another. It is possible to define for each population a 'neighbourhood', in such a way that an individual can only migrate to neighbouring populations.

In the *neighbourhood models* [17] the total population is conceived as evolving in a continuous inhabited area. Each individual moves in a bounded region and may interact only with those individuals living in the immediate locality, referred to as its neighbourhood. This model is highly distributed and fully exploits the available parallelism of the GA.

Acknowledgements

This text presents research results of the Belgian Incentive Program 'Information Technology' – Computer Science of the Future, and the Belgian programme on Interuniversitary Poles of Attraction, initiated by the Belgian State – Prime Minister's Service – Science Policy Office. The scientific responsibility is assumed by its authors. This research is supported also by Intel Supercomputer Systems Division, Beaverton, U.S.A. ('University Partners Program'), and the ESPRIT Parallel Computing Action of the E.C. (project 4065). D. Roose acknowledges the support of the A. von Humboldt Stiftung ; part of this work has been done during his stay as a Humboldt Research Fellow at the G.M.D., Sankt Augustin, Germany. R. Van Driessche is Research Assistant of the National Fund for Scientific Research (Belgium).

References

1. S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. Stat. Comput.*, 12(1):145–157, 1991.
2. L. Beernaert, D. Roose, R. Struijs, and H. Deconinck. A multigrid solver for the Euler equations on distributed memory parallel computers. *IMACS J. Appl. Num. Math.*, 7:379–393, 1991.
3. M.J. Berger. Adaptive mesh refinement for parallel processors. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 182–194. SIAM, Philadelphia, 1989.
4. J.E. Boillat. Load balancing and Poisson equation in a graph. *Concurrency : Practice and Experience*, 4(2):289–313, 1990.

5. L. Booker. Improving search in genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence, pages 61–73. Pitman, London, 1987.
6. A. Brandt. Multigrid techniques : Guide with applications to fluid dynamics. GMD Studien 85, GMD, St. Augustin, 1984.
7. N. P. Chrisochoides, E. N. Houstis, and C. E. Houstis. Geometry based mapping strategies for PDE computations. In *Proceedings of the ACM Supercomputing Conference '91*, pages 115–127, 1991.
8. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, 1989.
9. J. De Keyser and D. Roose. Load balancing data-parallel programs on distributed memory computers. Report TW 162, K. U. Leuven, Leuven, Belgium, Dec 1991.
10. J. De Keyser and D. Roose. Multigrid with solution-adaptive irregular grids on distributed memory computers. In G.R. Joubert, D.J. Evans, and H. Liddell, editors, *Proceedings of the International Conference on Parallel Computing '91.* Elsevier Science Publishers, B.V., 1991. to be published.
11. J. De Keyser and D. Roose. A software tool for load balanced adaptive multiple grids on distributed memory computers. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 122–128. IEEE Computer Society Press, 1991.
12. C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28:579–602, 1988.
13. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice-Hall, Englewood-Cliffs N.J., 1988.
14. G. C. Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 37–61. Springer Verlag, 1988.
15. G. C. Fox. A review of automatic load balancing and decomposition methods for the hypercube. In M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 63–76. Springer Verlag, 1988.
16. D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.
17. M. Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, number 496 in LNCS, pages 150–159. Springer-Verlag, Berlin, 1990.
18. J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence, pages 42–60. Pitman, London, 1987.
19. J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.
20. B. W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–308, 1970.
21. M.-H. Lallemand, H. Steve, and Dervieux A. Unstructured multigridding by volume agglomeration. Rapports de Recherche 1224, INRIA, Sophia Antipolis, May 1990.
22. S. McCormick. *Multilevel adaptive methods for partial differential equations.* SIAM, Philadelphia, 1989.
23. H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7:65–85, 1988.
24. A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl*, 11(3):430–452, 1990.
25. D. Quinlan and M. Lemke. Parallel local refinement based on hierarchical and asynchronous fast adaptive composite methods for distributed architectures. In *Proceedings of the Third European Conference on Multigrid Methods.* GMD, Sankt Augustin, 1991.
26. P. Sadayappan and F. Ercal. Nearest-neighbor mapping of finite-element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36(12):1408–1424, 1987.
27. P. Sadayappan, F. Ercal, and J. Ramanujam. Cluster partitioning approaches to mapping

parallel programs onto a hypercube. *Parallel Computing*, 13(1):1–16, 1990.

28. P. Sadayappan, F. Ercal, and J. Ramanujam. Parallel graph partitioning on a hypercube. In J. Gustafson, editor, *Proceedings of the 4th Distributed Memory Computing Conference*, pages 67–70. Golden Gate Enterprise, CA, 1990.
29. A. Schüller and G. Lonsdale. Parallel and vector aspects of a multigrid Navier-Stokes solver. *SIAM J. Sci. Stat. Comp.*, 1991. to be published.
30. H.T.M. Van der Maarel. Adaptive multigrid for the steady Euler equations. In S. McCormick, editor, *Proceedings of the 5th Copper Mountain Conference on Multigrid methods*, 1991. to be published.
31. R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3:457–481, 1991.
32. R.D. Williams. Supersonic fluid flow in parallel with an unstructured mesh. *Concurrency : Practice and Experience*, 1(1):51–62, 1989.

STRATEGIC DECISION ANALYSIS AND GROUP DECISION SUPPORT

SIMON FRENCH

*School of Computer Studies,
University of Leeds,
Leeds, LS2 9JT, UK*

Abstract. The context of group decision support within an organisation is discussed. In particular, the paper focuses on the needs of senior management teams facing long-term complex strategic issues. The use of facilitated workshops is discussed, particularly the format used by decision conferencing. Advances in computing power and displays have been central to the development of decision conferencing, because they allow the results of a decision analysis to be generated interactively and explored graphically. Some applications are noted to strategic decision making both within and outside the computer industry

Keywords. Computer supported co-operative work (CSCW) – Decision conferencing – Facilitated Workshops – Group decision support systems (GDSS) – Groupware – Strategic decision analysis

1. Introduction

Some four years ago I was appointed to the Chair of Operational Research and Information Systems at the University of Leeds. Given my meagre background in OR, I would have been fortunate to have been appointed to a chair in OR alone. Given my nonexistent background in information systems, to attach responsibility for that to my appointment was a considerable act of optimism on the University's part. Consequently, I have spent much of the past four years 'coming up to speed' in information systems theory. In many ways, this has not been too onerous a task: OR and information systems engineering have much in common. However, I have had much difficulty gaining any command of information systems terminology, particularly in the field of decision support.

Silver (1991) notes that historically a *decision support system (DSS)* has generally been defined as "a computer-based information system that supports people engaged in decision making activities". He goes on to give his own broader definition that a DSS "is a computer-based information system that affects or is intended to affect how people make decisions". Lewis (1991) surveys 39 commonly used introductory texts on information systems and found that three quarters relate their definition of information or information systems to decision making. Taking either of Silver's definitions, the study of DSSs would thus encompass the whole of the field of information systems.

Part of the problem is that, unlike many other academic disciplines, computer science and information systems theory seek to attach concepts to products, services and systems that someone is trying or will try to sell. Thus the clarity of academic definitions are soon sullied by their misuse in advertising and marketing. A spreadsheet is thought to sell better when marketed as a *personal decision support system*. You can probably only sell a database to the data processing manager, but call it an *executive information system (EIS)* and your potential customer base is anyone in management. At the moment, *group decision support system (GDSS)* seems to roll off the marketing people's tongues with some ease. If they can get the word *strategic* in, then the advertising blurb will be a winner!

My original intention had been to review different approaches to GDSS and their use in dealing with complex strategic issues, giving particular emphasis on decision conferencing. But with so many methodologies and software systems claiming to be GDSSs or, at least, components of GDSSs, the task has been beyond me. Instead I have chosen simply to concentrate on decision conferencing, but in doing so I have set its use within an organisational context which I hope may inform discussion of other approaches to group decision support (GDS).

Thus in the next section, I present a perspective of the organisational context of strategic decision making which derives from the work of Elliot Jaques. This perspective enables me to articulate some comments on the most appropriate places at which various types of DSS can help an organisation's decision making. In later sections I review approaches to decision analysis and the benefits that the growth in readily available computing power has brought. With this background, the decision conferencing approach to GDS is described and its applications reviewed.

2. Time-Span of Discretion and Strategic Decision Making

Group decision making, it seems to me, always takes place within the context of an organisation. It may be the trivial case in which the group itself is the organisation, such as a family deciding upon where to take their holidays. Or the group may be a committee within a university, a council of ministers within a government, a management team within a business, the board of directors in a multinational, or any one of many other examples. For definiteness, I shall set most of the following within a business context, but with suitable changes of vocabulary it applies to most, if not all organisations.

Over the past four decades, Elliot Jaques has developed an approach to the theory of organisations based on what he terms the *time-span of discretion*: see e.g. Jaques (1989). Figure 1 shows an example of a hierarchy that might form the basis of the organisation of a small firm. Jaques argues that the tasks and decision making undertaken at different levels in an organisation may be characterised by the time-span of discretion required. Directors may be typically working on strategies that will bear fruit in a 2-5 year timeframe. Their 'vision' is focused on this period, and they avoid dealing directly with the day-to-day operation of the firm. Line management focus their attention on a shorter timeframe, say 6 months to 2 years;

while the workers would focus their attention on the very short term, maybe 0-6 months. There are a number of points which should be made immediately.

Firstly, there may be more or less than three levels in the hierarchy: three are shown here only as an example. Secondly, the time-span of discretion at any particular level depends on the organisation's culture. In some, the workers may only have a time-span of discretion of perhaps a week; in others, maybe a year. Thirdly, the maximum time-span of discretion will vary between organisations. A small company may look at most five years ahead; a multinational, several decades. Fourthly, the level to which a people may (usefully) rise in an organisation is limited by their ability to envisage the future. I shall call this the limits of their vision. If someone is able only think up to two years ahead, then in the example of Figure 1 he or she should not rise above line management.

Of course, this is not to claim that people do rise no higher than the limits set by their vision. Jaques argues persuasively that organisations are best able to achieve their ends when accountability and responsibility flow through a hierarchy such as that in Figure 1. He terms such a structure an *accountability hierarchy (AH)*. When members of the organisation work at levels with time-span of discretion less than or equal to the limits of their vision, he terms the AH *requisite*. Building upon this, Jaques (1989) goes on to develop a normative theory of organisations.

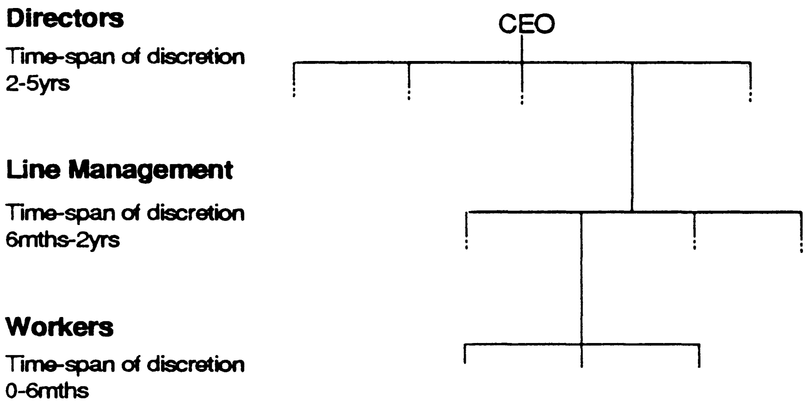


Figure 1: An organisational hierarchy for a small business

Descriptively, organisations may not match Jaques' model. People may have risen higher than their vision should allow. Matrix organisations (see, e.g., Handy, 1985) have no clear AH. At least they have no *single* hierarchy structuring accountability and responsibility within them. But each project team or task group within a matrix organisation may structure its working by means of an AH. Whatever the case, I shall take Jaques' conception of AHs to prescribe an, if not *the*, ideal structure for an organisation and set my discussion accordingly.

It is important to consider the dynamics of organisations: to think of organisations as processes as well as structures. Figure 2 suggests, at the level of gross generalisation, how the tasks undertaken at any particular level in an AH varies over time. Figure 3 sets this within the AH of Figure 1: but first concentrate on Figure 2. In the normal course of events, work consists of communicating and implementing strategy already decided upon, together with information gathering, monitoring and evaluating the performance of the current strategy. As time passes, the need for revision of the strategy will grow, maybe because of decisions taken at a higher level in the hierarchy. This will lead to a growth in the information gathering to investigate possible alternative strategies, and a declining emphasis on communication and implementation. When enough information has been gathered or when external factors dictate, there follows a short period of intense decision making, during which a new strategy is adopted. Next there follows a period in which the emphasis is on communicating and implementing this strategy; and the cycle repeats itself.

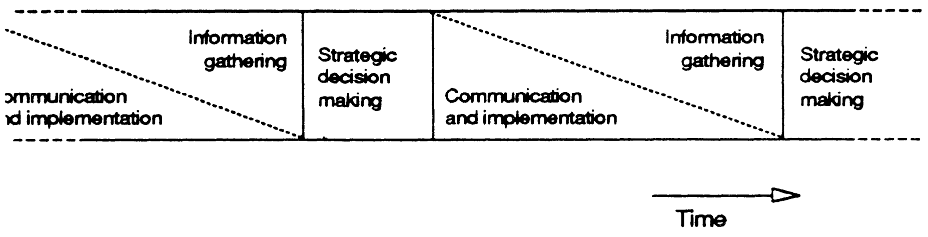

Information gathering
Strategic decision making
ommunication
d implementation
Information gathering
Communication and implementation
Strategic decision making
Time

Figure 2: Changes in mode of work over time at a level in a hierarchy

Embedding this view of the work at a level within an AH gives Figure 3. Note how the processes are staggered between the levels, as the decisions taken at one level cascade down the hierarchy, necessitating revisions of strategy at the lower levels.

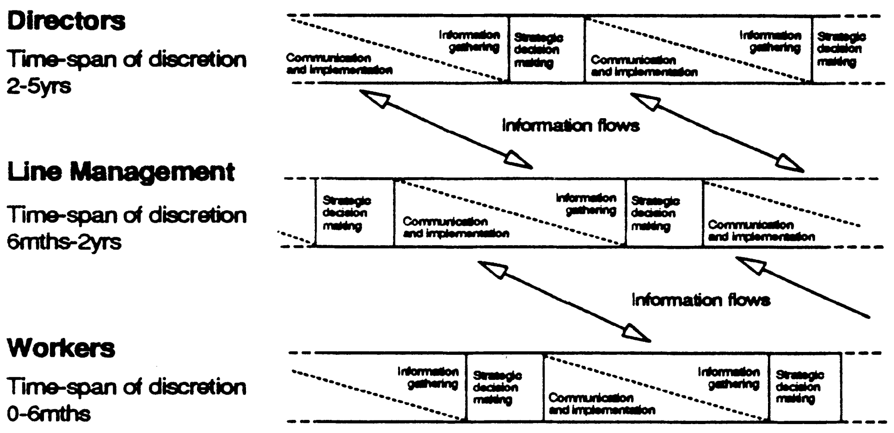

Directors
Time-span of discretion 2-5yrs
Information gathering
Strategic decision making
Communication and implementation
Information flows
Line Management
Time-span of discretion 6mths-2yrs
Workers
Time-span of discretion 0-6mths

Figure 3: Changes in mode of work over time in a hierarchy

I have used the term *strategy* fairly indiscriminately in the preceding paragraphs. A strategy broadly describes a course of action extending over a period of time, with guidance on how various contingencies are to be overcome, should they arise. It is common to contrast strategic decisions with tactical decisions. Strategies are contingency plans which guide the choice of appropriate actions as the future unfolds. Tactics are the possible actions that may be taken in the face of a particular contingency. Thus a strategy guides tactical decision making. The time-span of discretion concept leads to a further perspective on the difference between strategy and tactics. At any level within an AH, a strategy is broadly any policy or course of action that will have outcomes within the time-span of discretion at that level. Generally, tactics are actions that have their effect sooner than the lower limit of the time-span of discretion. Subject to the guidance implicit in a strategy, decisions on tactics may be left to lower levels in the organisation. Thus the tactics at one level are the strategies at lower levels.

This use, I am making, of the term *strategy* is very different to the conventional wisdom of categorising decision making as *strategic*, *tactical*, or *operational*, and I would not like to push my use too far. My purpose in taking the direction that I am is to emphasise that one of the qualities that determines how a person perceives and tackles a decision is whether its outcomes fall in his or her time-span of discretion.

Whether a decision is tactical depends as much on who makes it as on what the decision problem is. A company director would be perfectly capable of deciding how many paper clips to order for the next month, but he would not consider it a strategic decision. Lower in the organisation, a stationery clerk might consider such a decision strategic and ponder some time on it.

All the above ignores the fact that most organisations carry out a range of activities. For instance, a small manufacturing firm may have activities associated with production, marketing, finance and R & D. Each of these areas of activity will lead to a set of processes: see Figure 4. In a requisite organisation each 'copy of Figure 3' within Figure 4 would correspond with a different set of branches of the AH. In a matrix organisation, each copy would correspond to a particular task group. Note that the time between periods of strategic decision making will vary both within an area of activity and between areas of activity.

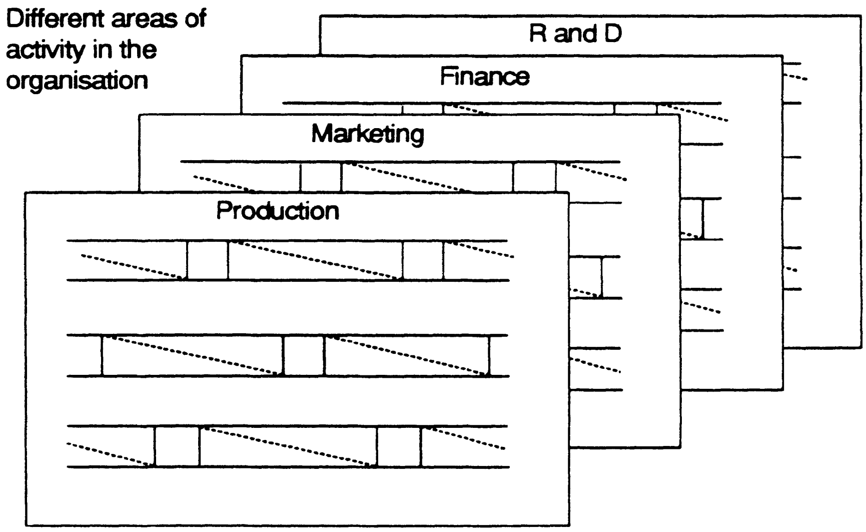


Figure 4: Different areas of activity within an organisation

3. Setting the context of a DSS within an AH

This perspective, simplistic though it be, provides a framework for the discussion of the appropriateness of different forms of computer-based support at different times and different levels within a organisation: see Figure 5. For instance, information gathering may be supported by straightforward database systems at lower levels, management information systems (MIS) at intermediate levels, and executive information systems at higher levels. All communication and implementation may be supported by *computer supported co-operative work (CSCW)* or *groupware* *systems*. Note that CSCW should provide the medium for communication both between different levels within an area of activity and between different areas of activity. In particular, it seems sensible to reserve DSS to describe systems which support the decision making phases.

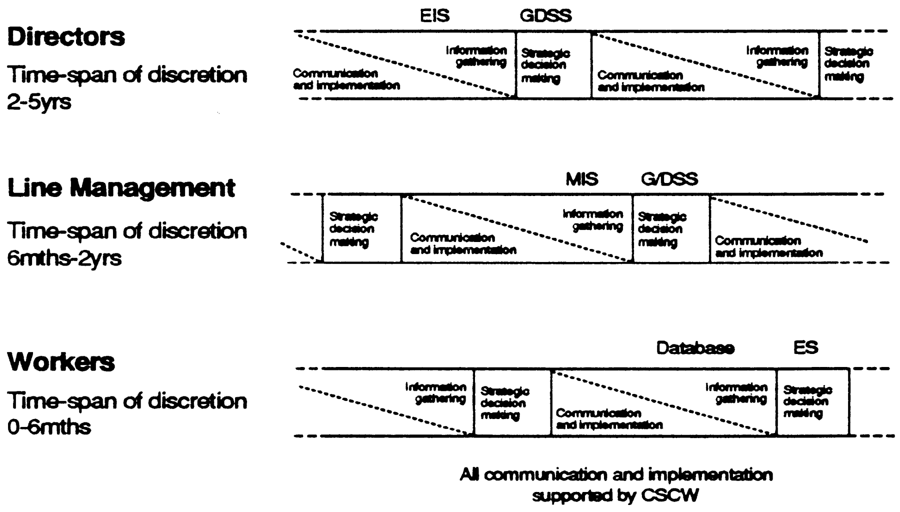

Directors
Time-span of discretion
2-5yrs
EIS
GDSS
Information gathering
Strategic decision making
Communication and implementation
Information gathering
Strategic decision making
Communication and implementation
Line Management
Time-span of discretion
6mths-2yrs
MIS
G/DSS
Strategic decision making
Communication and implementation
Information gathering
Strategic decision making
Communication and implementation
Workers
Time-span of discretion
0-6mths
Database
ES
Information gathering
Strategic decision making
Communication and implementation
Information gathering
Strategic decision making
All communication and implementation supported by CSCW

Figure 5: Using computer-based support within an organisation

The notion of time-span of discretion suggests that there may be different needs of computer-based support at different levels within an organisation. I doubt whether anyone will find this remark novel in the context of those systems which support information gathering. The attribution of database systems, MIS and EIS to the different levels in Figure 5 is entirely conventional. At the lowest levels raw or nearly raw data is needed to guide the day-to-day operation of the organisation. Higher up the summaries and analyses provided by an MIS are more appropriate to the detail needed to support tasks with a longer time-span of discretion. At the highest levels, gross summaries and comparative pictures of the organisation's performance are needed.

The ordering provided by time-span of discretion has a relationship with the structured-unstructured task dimension introduced by Gorry and Scott Morton (1971) or the programmed-nonprogrammed task dimension introduced earlier by Simon (1960). The longer the time-span of discretion of a task the more unstructured it tends to be, requiring more creativity and less programmed behaviour. The distinctive feature of using the time-span of discretion dimension is the mapping which Jaques suggests that it provides onto the organisational structure.

Turning to the types of DSS appropriate to the different levels in the organisation, one might make the following attributions. At the lowest levels with short time-span of discretion, decision making is more repetitive, structured and rule-based, suggesting that support may be most suitably provided by an expert system (ES) approach. Higher up the organisation, expert systems may not be appropriate, because the longer time-span of discretion the less it is possible to apply repetitive rules to help determine strategy. The successful applications of ESs reported in Ignizio (1991) support this assertion, although some proponents of ESs may not believe it.

DSSs that are appropriate to time-spans of discretion of a year or two will be much less structured that ESs. They may support individual decision makers or groups: hence I used 'G/DSS' in Figure 5. The sort of support I envisage here is that provided by the standard OR models, such as linear programming, augmented by systems that allow the user to express preferences and judgements. All the interactive multi-criteria methods surveyed by Belton (1990) provide suitable examples of appropriate DSSs.

At the top of the organisation, decisions are the least structured and have the longest time-span of discretion. They shape the overall strategy of an organisation. Taking decisions at this level requires much creativity and judgement, and flexible DSSs incorporating brainstorming and similar activities are necessary. They need to help evolve and shape judgement, especially in relation to explicit judgemental evaluation of alternatives. Moreover, typically at this level decision making is a group activity. Thus GDSSs are necessary, and ideally ones which help communication and the creation of a shared understanding between the group members. See also Phillips (1989). It is the support of this level of decision making activity within an organisation that I discuss below in Section 5, and I defer further discussion until then.

The communication and implementation activities within an organisation also need computer-based support. There is much current in the literature shaping the CSCW and groupware of tomorrow: see, e.g. the special issues of the *International Journal of Man-Machine Studies* of February and March 1991. The only comment I would make on CSCW is that it needs to support work at all levels in the organisation. The different time-spans of discretion at different levels may necessitate different interfaces at each levels to what is essentially the same system. Moreover, there is a potential difficulty in transferring results of empirical studies obtained on the efficacy of CSCW at one level to other levels. This will be significant for those seeking to demonstrate the value of CSCW. While it will be relatively easy to set up scientifically designed studies of CSCW at the lower levels, higher up the organisation it will be more difficult: CEOs do not have the time to play!

4. Bayesian Decision Analysis

I espouse the Bayesian approach to decision analysis which has grown from the founding work of Ramsay(1931) and Savage(1954). For a review of the theoretical underpinnings of this: see French (1986). More applied reviews may be found in

Clemen (1991), French (1989), Von Winterfeldt and Edwards (1986), and Watson and Buede (1987).

I mention the founding work of Savage because it is interesting to make an aside on the parallels between his *small worlds* concept and Jaques' time-span of discretion. Savage argued that an analysis is never carried out in the context of a full model of the real world, but rather a small world model. To build this small world the decision makers (DMs) abstract from the real world those aspects that affect their choice and ignore all aspects that are irrelevant to it. Phillips (1984) has developed this idea, though not explicitly from a starting point of Savage's work, into a theory of *requisite modelling*. In this Phillips describes the manner in which the analysis iterates through stages of more and more detailed model building until all that is (seems?) relevant to the decision is included. Thus he describes the process of building a small world.

This adds another perspective to Jaques' use of the time-span of discretion to characterise the mode of working at different levels in an organisation. For example, senior management need models from which they can omit or, rather, not build in particular details, thus capturing the spirit of remarks such as: "I don't know the details of how we will solve this one, but I am sure that Joe down in the production department will be able to. So it needn't detain us." Tactical decisions may be ignored in the model, because they relate to shorter time-spans of discretion appropriate to lower levels in the organisation. The higher one goes in an organisation the more gross the detail needed in the modelling. The small worlds of senior management are 'broad-brush', long-term, global pictures, whereas lower down an organisation more detailed local small worlds are required. These remarks further emphasise the need for DSSs to be tailored to the level in the organisation at which they will be used. The detail included in the models they use must be appropriate to the mode of working and time-span of discretion of their users.

It should be emphasised that (Bayesian) decision analysis is *normative*. It suggests how a decision should be made, not how an unaided decision is made. It is prescriptive, not descriptive. None the less, it is not dictatorial. Decision analysis helps the decision makers understand the opportunities and the choice before them. Enlightened by this understanding, they make the decision. See, e.g., Bell, Raiffa and Tversky (1988), French (1986, 1989), and Phillips (1984).

To provide this understanding, the assumptions underlying a decision model must be unambiguous, explicit and justified. For me, this means that I cannot recommend the adoption of several non-Bayesian families of models proposed in the literature: see French (1986). As you may imagine, however, there is far from universal agreement on what assumptions may be justified and hence which models may be used. Thus designers of DSSs would be well advised to survey the decision theory literature before incorporating a particular family of models into their system. I am afraid that information system theorists have an track record of re-inventing wheels, the roundness of which have been well investigated in other disciplines and found wanting.

Bayesian decision analysis has been with us in principle for some four decades, but its application within DSSs extends back over a far shorter period. Although the Bayesian decision models seem to have a simple structure, such as that of decision trees, and the calculations required by their analysis require little more than the four arithmetic operations, the number of computations necessary for realistically sized problems can be enormous. Thus analyses in the 1960s and 1970s required a mainframe, or at least a minicomputer. They were done 'offline' by consultants in major studies undertaken for an organisation. While the reports produced doubtless affected the organisation's high level decision making, they cannot really be taken to have constituted a DSS. Over the past decade, the growth of readily available microcomputing has changed this. Modern PCs are well able to support decision analyses (although full sensitivity analyses may require more power, perhaps that available through parallel processing environments hosted within a PC; Rios Insua and French, 1991). Moreover, graphical interfaces allow the results to be presented to and explored by the DMs in an informative way. Thus Bayesian methods are now providing the kernels of many DSSs. Clemen (1991) surveys some of the more common packages.

5. Facilitated Workshops and Decision Conferencing

The methods of GDSS which I wish to discuss further support strategic decision making at the highest levels within an organisation: see Figure 6. They primarily deal with matters requiring time-spans of discretion of more than a year. In other words, they are appropriate when substantial issues arise which will necessitate consideration of the organisation's overall strategy. Note the ellipse in Figure 6. It makes one point not made earlier. Any DSS supports elements of the information gathering prior to the decision and elements of the communication and implementation activities subsequent to the decision.

It has become the practice in some organisations, when complex strategic problems are faced, for an appropriate management team to meet together for a day or more to discuss and explore the issues (Eden and Radford, 1990). Often the teams are aided at these meetings by a *facilitator*. A facilitator is skilled in the process of group discussion, but seldom does he or she have any expertise in the context of the issues at hand, and even more seldom would he or she use such expertise in the discussion. The facilitator's role is to smooth the team's work, to help the process and make the team more productive, more creative. The content of the discussion comes entirely from the management team themselves. They are the specialists in their organisation's business. No one can know it as well as they do. Such meetings are known as *facilitated workshops* or as *decision conferences*, although the latter term was until recently reserved for the style of facilitated workshop developed by Cam Peterson, Larry Phillips and their colleagues.

More and more these, decision conferences are being supported by computer-based tools. Using a projected word processor to capture a mission statement or action list is very effective. Seeing the words appear and be edited on the screen seems to provide a much stronger focus for discussion than flipcharts and whiteboards can provide. Cognitive modelling techniques (Rosenhead, 1989), now embodied in the

COPE software package, help the DMs explore the issues and concerns which brought them together. Decision analytic software enables the DMs to investigate the implications of their beliefs and their preferences. More importantly, the models are implemented in such a way that variations in the input judgements can be explored easily. Thus the team can discover both the importance of differences in opinions between the team members and the sensitivity of their conclusions to these differences and to those judgements of which they are most unsure.

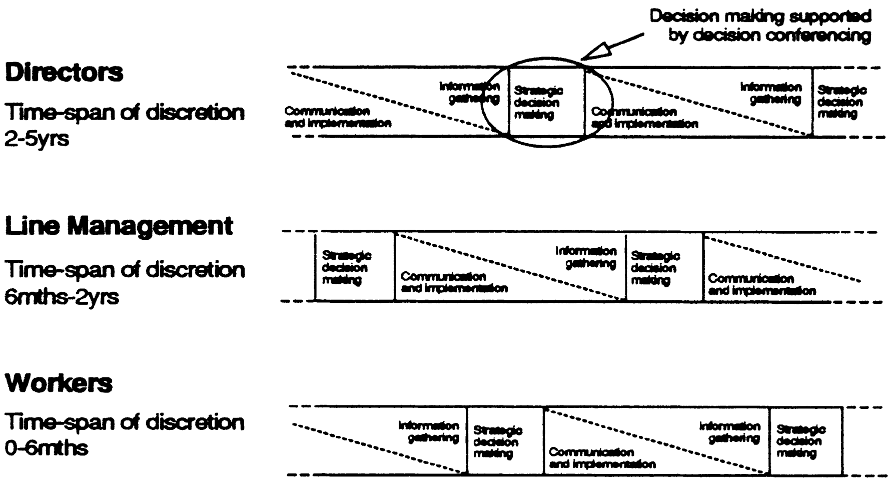

Decision making supported by decision conferencing
Directors
Time-span of discretion 2-5yrs
Information gathering
Strategic decision making
Communication and implementation
Line Management
Time-span of discretion 6mths-2yrs
Workers
Time-span of discretion 0-6mths

Figure 6: Decision conferencing provides GDSS for higher levels within an organisation

A decision conference is generally a two-day event. Other timescales are possible; but the inclusion of a night is more or less essential. In the evening the DMs are able to relax together and reflect on the progress and discussion so far. This reflection, together with the distance from the previous day's deliberations that a night's sleep brings, helps the team acquire a more mature perspective on the issues that concern them. Without the night's break the team may have 'second thoughts' soon after the conference ends, perhaps on the journey home, and much of the value of the event will be dissipated as their commitment to its conclusions evaporates.

The entire management team take part in the conference, which concentrates entirely upon the strategic issues that led to it being called. There are no time-outs to consider peripheral matters 'while the team are together'. For that reason it is sensible to hold a decision conference away from the team's normal place(s) of work: perhaps a country hotel or a purpose built decision conferencing suite. ICL, who have been at the forefront of developing decision conferencing as a management tool, have built a decision conferencing suite, in which all the necessary information technology is carefully and unobtrusively installed. They also have a number of *pods*, in which the same information technology is present, but, far from being unobtrusive, a virtue is made of its visibility. The computer-based GDSS facility at the University of Arizona provides another style of environment (Nunamaker *et al*, 1988). The relative advantages of the ICL facilities and the Arizona facility are discussed by Phillips (1989).

The facilitator leads the conference, guiding the discussion forward in a constructive fashion. He or she is expert in three areas: group dynamics; rational decision theory; and communication. The expertise in group dynamics enables the facilitator to involve all participants in the debate. He or she contributes to the decision process by ensuring that the objectives and uncertainties are taken account of in a rational manner and by keeping the team task oriented. Throughout the facilitator uses communication skills to ensure that all participants understand all the issues, as they are identified. The facilitator is assisted by an *analyst* and, possibly, a *recorder*. The analyst runs the software, generating models of the issues as they arise, which help the management team gain insight into the situation facing them. The recorder uses a wordprocessor to record the development of the debate and the reasoning behind the judgements and decisions made by the management team. Because of the presence of the recorder, the team are able to take a record of all the important conclusions and an action list with them at the conclusion of the conference. More and more, the roles of recorder and analyst are becoming identified. In the early days of decision conferencing, the recorder and analyst needed a computer each; but with the advent of multi-tasking windowing environments it is possible for one person to fulfil both roles. Moreover, there are advantages if one person does. Inevitably, an analyst is far more closely involved with the process than a recorder and so better placed to record, for instance, the reasoning underlying a particular model.

Each decision conference is different. It evolves according to the needs of the DMs and not according to some fixed agenda. There are, however, common themes and patterns. The facilitator is always careful to ensure that the opening discussion is as wide-ranging as possible. It is a rare decision conference in which a single focus for discussion emerges in the opening few minutes. Throughout the event, discussion returns again and again to the main issues as insights are gained and understanding shared. No model is taken as definitive: all are simply vehicles for exploring ideas. Surprisingly, the analysis in decision conferences needs much less hard data than one would, at first, think. Strategies have to be costed: that is clear. But the costings need only be rough. It is a broad brush picture that the event seeks to create. Detail can be added at a later date. (c.f. the remarks above about 'small world' models)

Decision conferences are highly creative events. Typically, participants arrive as individuals, bemused and uncomfortable, unsure of a way forward. During the event they create, evaluate, modify and re-evaluate options, building a strategy which they all support. They leave as a committed team with a common purpose and understanding of the issues, ready to implement the strategy they have created.

Decision conferences have found many applications. During the mid-1980s ICL incorporated the approach into its organisational culture (Hall,1986). Decision conferences were used to shape ICL's strategy in many fields, from R &D decisions concerning hardware development to marketing strategy and consultancy. Many other companies have used decision conferencing, e.g. Dolland and Aitcheson, Mars and Pactel, in areas as diverse as developing strategic options, allocating budgets between the various divisions of an organisation, formulating organisational change, developing industrial relations strategies, evaluating competitive bids, and choosing investment strategies. Belton (1985) describes the use of decision conferencing (although she explicitly avoids the term) to the choice of a computerised financial management system. The method has been used in local and national government. Indeed, the *strategic choice approach*, which may be looked upon as a form of decision conferencing, grew out of early work in facilitating local government decisions (Friend and Hickling, 1987). Recently, decision conferencing was used to investigate the decision making after the Chernobyl accident in the Soviet Union (French *et al*, 1992). While this last application was not a full use of the technique as a GDSS – quite deliberately no decisions were taken during the conferences – it does point the way to use of the technique in dealing with radiation protection issues.

Acknowledgements

I am especially grateful to Larry Phillips for many discussions and much advice and help. I have also had many helpful discussions with Valerie Belton, Peter Hall, David Rios Insua, Nigel Tout and Graham Wharton.

References

Bell, D.E., Raiffa, H. and Tversky, A. (eds): 1988, *Decision Making*, Cambridge University Press, Cambridge

Belton, V.: 1985, 'The use of a simple multi-criteria model to assist in selection from a shortlist', *Journal of the Operational Research Society*, **36**, 265-274

Belton, V.: 1990, in Hendry, L.C. and Eglese, R.W., eds., *Operational Research Tutorial Papers 1990*, Operational Research Society, Birmingham, pp. 53-101

Clemen, R.T.: 1991, *Making Hard Decisions*, PWS-Kent, Boston

Eden, C. and Radford, J. eds.: 1990, *Tackling Stategic Problems*, *Sage*, London

French, S.: 1986, *Decision Theory*, Ellis Horwood, Chichester

French, S. ed.: 1989, *Readings in Decision Analysis*, Chapman and Hall, London

French, S., Kelly, G.N. and Morrey, M.: 1992, 'Decision conferencing and the international Chernobyl project', *International Journal of Radiation Protection*, in press

Friend, J. and Hickling, A.: 1987, *Planning under Pressure*, Pergamon, Oxford

Gorry, A.G. and Scott Morton, M.S.: 1971, 'A framework for management information systems', *Sloan Management Review*, **13**, 55-70

Hall, P.: 1986, 'Managing change and gaining corporate commitment', *ICL Technical Journal*, **7**, 213-227

Handy, C.: 1985, *Understanding Organisations*, 3rd Edn, Penguin, Harmondsworth

Ignizio, J.P.: 1991, *Introduction to Expert Systems*, McGraw-Hill, New York

Jaques, E.: 1989, *Requisite Organisation*, Cason Hall, Arlington

Lewis, P.J.: 1991, 'The decision making basis for information systems: the contribution of Vickers' concept of appreciation to a soft systems perspective', *European Journal of Information Systems*, **1**, 33-43

Nunamaker, J.F., Applegate, L.M. and Konsynski, B.R.: 1988, 'Computer-aided deliberation: model management and group decision support', *Operations Research*, **36**, 826-848

Phillips, L.D.: 1984, 'A theory of requisite decision models', *Acta Psychologica*, **56**, 29-48

Phillips, L.D.: 1989, 'Decision conferences: description, analysis and implications for group decision support', DAU Technical Report, 89-2, London School of Economics

Ramsay, F.P.: 1931, *The Foundations of Mathematics and other Logical Essays*, Keegan Paul, London
Rios Insua, D. and French, S.: 1991, 'A framework for sensitivity analysis in discrete multi-objective decision-making' *European Journal of Operational Research*, **54**, 176-190
Rosenhead, J. ed.: 1989, *Rational Analysis for a Problematic World*, Wiley, Chichester
Savage, L.J.: 1954, *The Foundations of Statistics*, Wiley, New York
Silver, M.S.: 1991, *Systems that Support Decision Makers*, Wiley, Chichester
Simon, H.A.: 1960, *The New Science of Management Decision*, Harper and Row, New York
Von Winterfeldt, D. and Edwards, W.: 1986, *Decision Analysis and Behavioural Research*, Cambridge University Press, Cambridge
Watson, S.R. and Buede, D.M.: 1987, *Decision Synthesis*, Cambridge University Press, Cambridge

TEN YEARS OF ADVANCES IN MACHINE LEARNING

Y. KODRATOFF

CNRS & Université Paris Sud, LRI, Bldg 490, 91405 Orsay, France

Abstract. Inference is a very general reasoning process that allows to draw consequences from some body of knowledge. Machine learning (ML) uses the three kinds of possible inferences: deductive, inductive, and analogical. We describe here different methods, using these inferences, that have been created during the last decade to improve the way machines can learn. We include also genetic algorithms as an induction technique. We restrict our presentation to the symbolic aspects of connectionism.
Due to space limitations some of the entries of this paper will be empty, especially those that would deal of well-known techniques. The interested reader will consult "The Machine Learning Series" cited at the beginning of the references.

1 - Introduction and definitions

Inference is a very general reasoning process which covers many particular kinds. Let us note A $\Vdash$ B that one can infer B from A. For any inference rule A $\Vdash$ B, $\Vdash$ is said to be **truth-preserving** if whenever A is TRUE, then B is also TRUE.
Deduction is classically defined as the mode of inference that preserves truth, i.e., inference A $\Vdash$ B, is a deduction if and only if B is TRUE whenever A is TRUE. Induction does not preserve the truth, and, in order to somewhat clarify the problem of induction, we shall give, in section 3, an explicit description of several inductive processes.
Analogy does not preserve truth either. Its difference with induction is that it relies on the comparison between existing complete knowledge and a set of knowledge to be completed. For instance, in Gentner's Structure-mapping theory, "the central intuition is that an analogy is the mapping of knowledge from one domain (the base) into another (the target) which conveys that a system of relations known to hold in the base also holds in the target" (Falkenhainer, Forbus, and Gentner, 1986).
ML makes uses of the three types of reasoning. Associated to each type, we find different systems. Some systems, such as Carbonell's group PRODIGY, use the combination of several of them. Figure 1, below, gives an overview of the different techniques, and shows how they rely on the way the inferences are drawn.
The main bulk of ML systems belongs to induction. To these techniques, we added the more recent ones, connectionism and genetic algorithms, even

though they are not born within the community of ML.

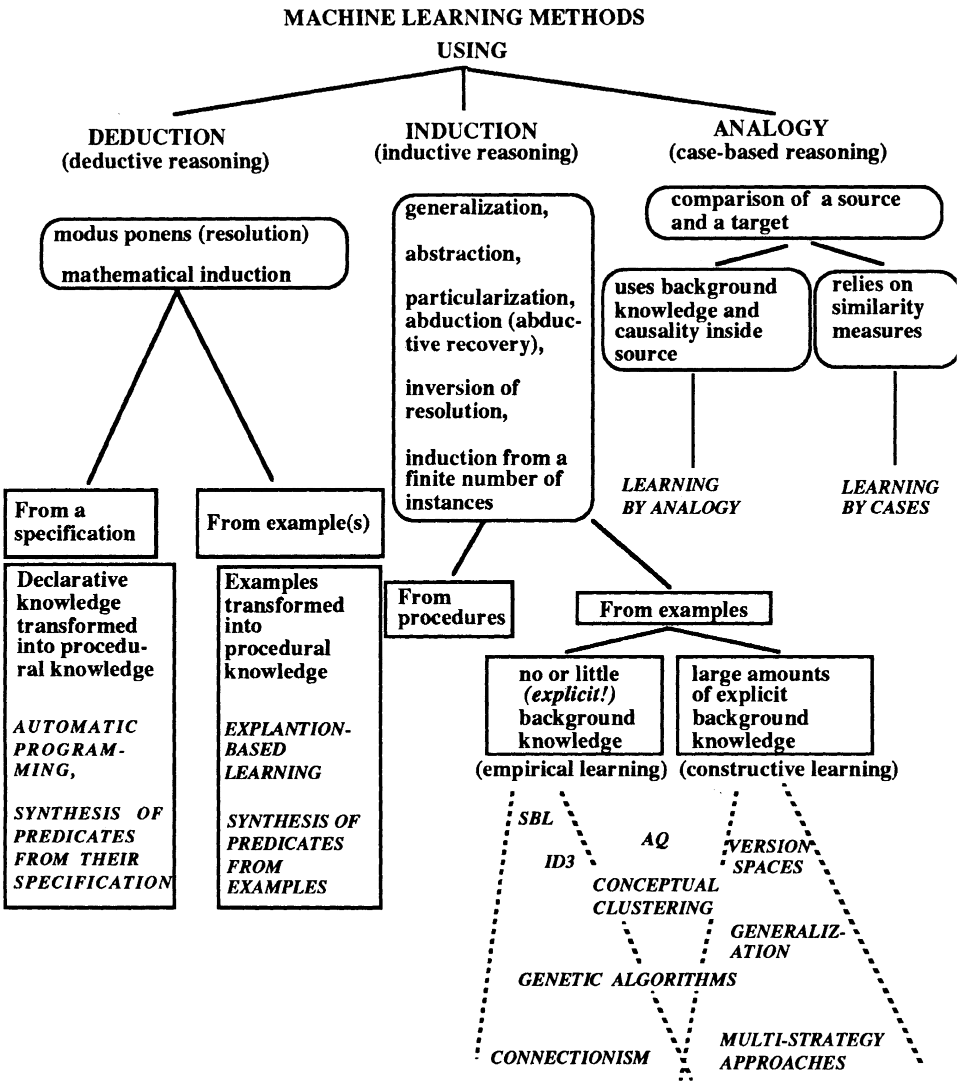

MACHINE LEARNING METHODS
USING
DEDUCTION
(deductive reasoning)
INDUCTION
(inductive reasoning)
ANALOGY
(case-based reasoning)
modus ponens (resolution)
mathematical induction
generalization,
abstraction,
particularization, abduction (abductive recovery),
inversion of resolution,
induction from a finite number of instances
comparison of a source and a target
uses background knowledge and causality inside source
relies on similarity measures
LEARNING BY ANALOGY
LEARNING BY CASES
From a specification
From example(s)
Declarative knowledge transformed into procedural knowledge
AUTOMATIC PROGRAMMING,
SYNTHESIS OF PREDICATES FROM THEIR SPECIFICATION
Examples transformed into procedural knowledge
EXPLANTION-BASED LEARNING
SYNTHESIS OF PREDICATES FROM EXAMPLES
From procedures
From examples
no or little (explicit!) background knowledge
large amounts of explicit background knowledge
(empirical learning)
(constructive learning)
SBL
ID3
AQ
VERSION SPACES
CONCEPTUAL CLUSTERING
GENERALIZATION
GENETIC ALGORITHMS
CONNECTIONISM
MULTI-STRATEGY APPROACHES

Figure 1. ML methods using three types of inferences.

Some methods, such as ID3, follow the widely acknowledged opinion that induction is knowledge poor. As stated in figure 1, however, this means poor-in-explicit-knowledge. A large amount of background knowledge is actually contained in the way examples are represented. On another hand, many inductive systems use explicit background knowledge in order to guide their

inductive leaps.
In the following all the systems present in the leaves of figure 1 will be detailed, except induction from procedures that yielded no large system yet.

2 - Systems based on deduction

2.1 - Automatic programming

Automatic programming (AP) developed mainly outside the main stream of ML. Let us give a somewhat sketchy description of what it is about.

- INTUITIVE PRESENTATION

Informally speaking, let us suppose that you have a full refrigerator and you would like a tasty and nice sandwich composed from things that are in this refrigerator. Therefore, you know what you have (refrigerator) and what you would like to have (tasteful and nice sandwich). You can now imagine a robot which would prepare such a sandwich for you.

The goal of AP is to construct automatically programs which make possible to the robot to compose your sandwich disposing with your refrigerator and knowing only your requirement (tasteful and nice) on this sandwich. Your requirement describes what the robot has to achieve, but it does not describe how this particular goal has to be achieved. In other words, your requirement is not a procedure which describes step by step what the robot has to do in order to achieve the desired goal. AP aims at providing such a procedure.

A bit more formally, your specification of the program *make_sandwich* consists of an input x (things that are in refrigerator), an input condition P(x) (for instance, these things must be edible, written *edible(x)*), an output z (*sandwich(z)*), and an input-output relation Q(x,z) (*composed_from(x,z) & tasty(z) & nice(z)*). The goal of AP is to construct automatically the program *make_sandwich*, i.e., construct automatically a procedure which describes step by step what the robot has to do in order to achieve the desired goal.

- FORMAL PRESENTATION

Formally now, a specification of a program Prog consists of an input vector x, an input condition P(x), an output vector z, and an input-output relation Q(x,z). The input condition P(x) is TRUE, if and only if x is an acceptable input for the desired program. The input-output relation Q(x,z) describes relational[1] links between the input and the output. In other words, AP tries to

[1] By relational links we mean here that it is not necessary to describe in which way we will

find a program Prog such that
(R1) $z = \mathrm{Prog}(x)$
(R2) if $P(x)$ then $Q(x,z)$.

For instance, the quotient-remainder problem is specified as follows:

the input vector x is the couple	(x_1, x_2)
the input condition P(x) is the formula	$x_2 > 0$
the output vector x is the coupl0e	(z_1, z_2)
the input-output relation Q(x,z) is the formula	$(x_1 = x_2 * z_1 + z_2) \& (z_2 < x_2)$

When P and Q are formulae, as it is in this example, the specification of Prog by (R1) and (R2) is called a **formal specification**. However, there are other ways to specify a program. For instance, one may give a finite set of input-output values, as it is the case in program synthesis from examples. One may also give a trace of the execution of a desired program. In this case, we speak of program synthesis from traces.

When a program is specified by a formal specification, AP obtains Prog through an inductive proof[2] of the so-called **Specification Theorem**

$$\forall x\ (P(x) \Rightarrow \exists z\ Q(x,z)), \qquad \text{(ST)}$$

This theorem comes simply from the specification "*for x satisfying P(x), find z verifying Q(x,z)*". We quantify here universally the input vector x and existentially the output vector z. Let us denote by F(x) the formula $P(x) \Rightarrow \exists z\ Q(x,z)$. Then, ST is exactly $\forall x\ F(x)$. Let ϕ denotes the Skolem function corresponding to ST, i.e., $z = \phi(x)$. When the input vector x belongs to a well-founded domain, from an inductive proof of ST it is possible to extract a recursive definition of ϕ. In other words, an inductive proof of ST provides a recursive definition (i.e., a program) for ϕ. The function ϕ, obtained in this way, corresponds to the target program Prog. Because the program obtained is extracted from a constructive proof of ST, its correctness with respect to the specification is assured (Sato, 1979). In other words, the proof of ST becomes as well a proof for the formula $\forall x\ (P(x) \Rightarrow Q(x,\mathrm{Prog}(x)))$.

This explains the importance of inductive theorem proving for automatic construction of programs from formal specifications. However, the construction of systems which perform induction proofs for theorems containing existential quantifiers is not a simple task.

---

compute the output z for a given value of the input x.

[2] We call by inductive proof any transformation of a given theorem, which uses explicitly or implicitly the induction principle.

Program synthesis from input-output examples and from computation traces are described in detail in (Kodratoff, 1988).

2.2 - Predicate Synthesis from their specifications

- INTUITIVE PRESENTATION

Let us consider the following specification for the subtraction function in natural numbers: *Given two natural numbers x and y, find z such that x + z = y*. To this specification corresponds the so-called specification theorem (Manna and Waldinger, 1980; Franova, 1988)

$$\forall x \, \forall y \, \exists z \, (x + z = y) \qquad (T_1)$$

For instance, for x=3 and y=7 we have z=4. Let us consider now x=7 and y=3. In the theory of natural numbers with the following recursive definition of addition +: $\mathcal{NAT} \times \mathcal{NAT} \rightarrow \mathcal{NAT}$

ai26: $\mathbf{0} + u = u.$ ai28: $u + \mathbf{s}(v) = \mathbf{s}(u + v).$

ai27: $u + \mathbf{0} = u.$ ai29: $\mathbf{s}(u) + v = \mathbf{s}(u + v).$

obviously, there is no z verifying 7+z=4. Therefore, T_1 is FALSE.

We can see, however, that there is a set U of couples (x,y), for which we can find z satisfying the input-output relation x+z=y. For instance, the couples (3,7), (1,8), (25,40) belong to U, because the respective outputs are 4, 7 and 15. This set U can be characterized by a predicate P, which, when taken as a precondition to T_1 makes TRUE the new theorem

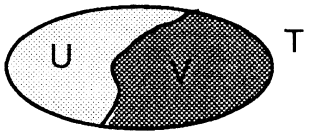
$$\forall x \, \forall y \, \{P(x,y) \Rightarrow \exists z \, (x + z = y)\}. \qquad (T_2)$$

The goal of predicate synthesis from formal specification is to provide a (recursive) definition for P.

- SEMI-INTUITIVE PRESENTATION

More formally, let us suppose we want to prove a theorem $\forall x \, A(x)$ in a set T. Let us suppose we have found out this theorem is FALSE, but A(x) is verified for some x. We say that $\forall x \, A(x)$ is **partially** FALSE. Since this theorem is universally quantified, to say that the theorem is FALSE, means that there are elements y in T for which $\neg A(y)$ holds.

Therefore, we can write T as a union of two sets, one, say U, to which all elements satisfying A belong, the other, say V, to which all elements satisfying $\neg A$ belong.



The problem is to characterize, by a predicate P, the set U in such a way that if an element x of T satisfies this P(x) then $x \in U$, i.e., A(x), also is satisfied. Thus, the theorem $\forall x \, \{P(x) \Rightarrow A(x)\}$ is then TRUE.
To look for a characterization of U corresponds to a search for preconditions, a well-known problem in automatic programming from incomplete specifications (Guiho, 1983; Smith, 1985).

- FORMAL PRESENTATION

Let $\forall x\ A(x)$ be a partially FALSE theorem. Let P ($\neq$ FALSE) be a predicate such that $\forall x\ \{P(x) \Rightarrow A(x)\}$ is TRUE. We say that the theorem $\forall x\ A(x)$ is a **formal specification** of the **predicate** P, and we call **Predicate Synthesis from formal specifications** (PreS) the task of obtaining P from its formal specification.

2.3 - Explanation-based learning

We already described EBL in several introductory courses (Kodratoff, 1988). In principle, EBL transforms a general rule Rg1 into a more particular Rp. Rp is more efficient than Rg1 because it contains implicitly the analysis of the application of Rg1 to a particular example. Therefore, EBL is a way to acquire some strategic knowledge on the way to use Rg1. This knowledge is directly incorporated in Rp. Since Rg1 and Rp are written in the same logic, no meta-rule is added to the set {Rg} of general rules one started from. In standard EBL, {Rg) and {Rp} belong to first order logic. PRODIGY contains, besides a set of first order {Rg1}, also some meta-knowledge on how the problem solver works, thus some second order knowledge. Combining these two sources of knowledge, it is therefore able to generate both rules {Rp} in first order, and rules {Rp2} in second order, that tell how to use efficiently the rules {Rg} or {Rp}. For instance, PRODIGY can generate rules like "Select IS_POLISHED(<obj>) before IS_REFLECTIVE(<obj>) if both are present in the same conjunctive subgoal" (Carbonell and Gill, 1990).

2.4 - Synthesis of predicates from examples

Empty due to space limitation. See (Muggleton and Buntine, 1988; Rouveirol, 1991; Quinlan, 1991).

2.5 - Explanation-Based Learning (EBL) in strong theory domains

Empty due to space limitation. See, however, Kodratoff (1988, 1989, 1990a, 1992).

2.6 - Explanation-Based Learning (EBL) in weak theory domains

Empty due to space limitation. See, however, Kodratoff (1988, 1989, 1990a, 1992).

2.7 - PAC-learning

The most popular way to transform inductive procedures into sounder ones is the measurement of the statistical amount of errors that are possible with a

given probability. This approach is taken by Valiant's (1984) probably approximately correct (PAC) learning.
The problem stated by Valiant is the characterization of learning problems that can be learned in polynomial time, with an approximation of the probable error on the result of the learning. In order to illustrate the concept, consider the example of a new casino entrance watcher. He will have to learn to recognize from sight people that are under 18. He will learn this task during a training phase, by observing an experienced watcher in action. His employers do not want him to take too much time to learn an efficient recognition procedure, hence the polynomial time requirement. They will want to have some confidence in his judgments, hence a confidence factor δ. Suppose that the employers want him to identify non adults correctly in 99.9 % of the cases, thus $\delta = 0.001$. They allow nevertheless allow some tolerance ε on the errors he will be doing, say, he should not bother for nothing more than one adult over 10 000. The error tolerance will thus be $\varepsilon = 0.0001$. In other words, the employers want to be sure, with a probability greater than ε, to have a bad choice with a probability less than δ.
More formally, let U be a countable universe of objects, the set of all persons gambling at casinos, in our example. Let L_1, L_2, etc., be a countable family of subsets of U (subsets of such gamblers). The task is is to identify one of these subsets L_{un}, given access to a sampling oracle ORA (the set of people met by our new watcher during his training phase). In our example, L_{un} is the set of people under 18 of age, and ORA is the experienced watcher who is working before the inexperienced one. Let us suppose that we dispose of a distance d that defines the distance between two subsets L_i and L_j. We shall say that an identification procedure that learns to recognize L_h is a PAC correct identification of L_{un} if and only if we have a probability less than δ that the distance between L_{un} and L_h is greater than ε, in other words that

$$\Pr[d(L_{un}, L_h) \geq \varepsilon] \leq \delta.$$

To make this definition totally correct, we still need to define d. Let D be a (possibly unknown) probability distribution on the elements of U. Let DIFSYM(S, T) be the symmetric difference of sets S and T, i.e., the sets of elements that are in S or in T, but not in both, DIFSYM(S, T) = (S - T) $\cup$ (T - S). Let $\Pr_D(x)$ denote the probability (relative to the distribution D) to find the element x in U. Then define

$$d(S, T) = \sum_{x \in \mathrm{DIFSYM}(S, T)} \Pr_D(x)$$

This distance is the probability that a random call will draw an element which is either in S or T, and in none other subset of U. It is the distance one has to choose in order to define precisely a PAC identification.

In order to illustrate these definitions, let us present an example from Angluin

and Laird (1988). Let us consider the problem of learning a rule that agrees with m examples, from a set of n given rules. Let $L = \{L_1, ..., L_n\}$ be this set of n rules. An algorithm that needs $m = (1/\varepsilon) \ln(n/\delta)$ examples to learn such a rule can be said to be a PAC learning algorithm. For this, let us show that a rule agreeing with at least m examples shows an error greater than ε only with probability less than δ. Let L_h be a rule with error such that $d(L_{un}, L_h) \geq \varepsilon$. The probability for a random example to agree with L_h is less than $(1 - \varepsilon)$; thus the probability for m random examples to all agree with L_h is less than $(1 - \varepsilon)^m$, which is bounded by $e^{-\varepsilon m} = \delta/n$. Obviously, there are at most N-1 rules that give such a bad answer, thus the probability that one of them agrees with all m examples is bounded by δ. Thus, by definition, our algorithm is a PAC learning algorithm.
Suppose that we are in the simple case where only two rules are possible, then the number of examples necessary to learn with an error less than 0.0001 with a confidence greater than 99% is thus $m = 10\ 000 \ln (200) \approx 53\ 000$. Since most people will need little more than one example to do the job, one could question the links of PAC learning with reality.
However, in a very recent work, Greiner and Elkan (1991) compute the number of examples to build a representation, and find bounds of the same order as those of PAC learning. In other words, this shows that the above 53 000 examples are actually needed for establishing a representation of the examples in which the rules make sense, besides the simple learning. It is then quite reasonable to ask so many examples for a task as complex as building a representation and learning in this representation. In such a case, many of the examples are implicitly contained in the knowledge about the domain when it is introduced in the representation.
This allows to understand two things. First, why simple knowledge-poor systems do learn anything at all from relatively few examples? Because they use a representation of knowledge which contains already large amounts of (implicit) examples. Secondly, one of the main trends of ML research during these last 10 years has been to combine examples to existing knowledge in order to refine the learning. This knowledge, now explicitly provided, replaces the masses of examples that would be necessary without it.

3 - A description of some inductive processes

We shall refine Peirce's definition (1965) of induction by seeing it as being built from elementary inductive processes, such as abduction, and several other processes described later in this section. For us, induction will be "the process by which one builds theories from facts," as stated by Peirce, and which combines some of the elementary inductive processes with as many deductive steps as deemed necessary. In that sense, induction is defined as a

multi-strategy mechanism.
The best known elementary inductive processes used in machine learning are **generalization** and **abduction** (Michalski and Kodratoff, 1990). Without any pretense to be exhaustive[3], we shall present here five other forms of inductive reasoning that we shall call **causality determination**, **attribution of properties**, **spatial or temporal inclusion**, **clustering & chunking**, and their relation to **numeric induction**.

3.1- Abduction

Let us recall that the classical rule for abduction, as given by Peirce, is an inductive version of *modus ponens*, i.e., a kind of inversion of *modus ponens* in which instead of deducing B from A and $A \Rightarrow B$, one induces A from B and $A \Rightarrow B$[4]. For instance, knowing that $\forall x[\text{Drunk}(x) \Rightarrow \text{Stumbling}(x)]$, one deduces from Drunk(Bob) that Stumbling(Bob). On the contrary, one induces from seeing a particular person stumbling that he might be drunk.

3.2 - Generalization

Generalization is also a process by which modus ponens is inverted, but the implication has the form $\forall x\ [F(x)] \Rightarrow F(a)$. This implication is always TRUE, therefore modus ponens is simply "from $\forall x\ [F(x)]$, infer $F(a)$." Generalization thus infers $\forall x\ [F(x)]$ from $F(a)$.
In practice, generalization is very important in the organization of knowledge, it builds trees of dependencies the links of which are IS_A links. Many details about generalization can be found in Michalski (1983, 1984), Dietterich and Michalski (1981), Kodratoff (1983, 1988, 1990a), Vrain (1990).

3.3 - Causality determination

When several phenomena seem to occur simultaneously, one might hypothesize that some are the cause of the others. Choosing which phenomenon causes the other can be done by deduction, and in that case, we dispose of a proof of what the cause is, but it can be also done by induction. Unfortunately, we have no place here to insist on the interesting problem of

[3] For instance, we do not study here these types of inductive inferences one can get by using a sub-part of (albeit its name, deductive) mathematical induction, such as inducing on a finite number of instances, inducing on incomplete base cases study, etc.

[4] An important application of induction and of abduction is the completion of failed proofs (Cox and Pietrzykowski, 1986; Duval and Kodratoff, 1990). Abduction is used as follows. A proof fails because one fails to prove B, it is noted that $A \Rightarrow B$, and one makes the abduction of A, which of course proves B, and therefore allows to complete the proof.

probabilistic causality, see, for instance, (Dupré, 1990).
This structures knowledge by creating semantic links "CAUSES" inside the knowledge.

3.4 - Attribution of properties

When several simultaneous observations are done, one can also hypothesize that one of them is a concept, while the others are the properties of this concept. Choosing which is the concept, and which are the properties, is an induction. It creates "HAS_PROPERTY" links among the knowledge.

3.5 - Spatial or temporal inclusion

Spatial (or temporal) inclusion means that all events happening within the spatially (or temporally) included region implies that it happens as well in the including one. This information structures the knowledge in PART_OF trees of dependency, instead of the IS_A ones built by the generalization process.

3.6 - Clustering and chunking

Clustering brings together several pieces of knowledge, such as objects of the world, concepts, properties, or relations. For a thorough discussion of the links between symbolic and numeric approaches, one can consult Stepp and Michalski (1983), and Kodratoff and Diday (1991).
Chunking keeps as a unit a sequence of operations which has proven useful during a problem solving episode. This approach as been exemplified by the systems SOAR (Laird et al. 1986, 1987) which implements chunking in a problem solving environment and PRODIGY (Carbonell, Knoblock, and Minton, 1990) which does so among other mechanisms.

3.7 - Numeric induction

Let us present here the definition that can be obtained through a definition of probabilistic causality, as stated by (Suppes, 1984), for instance. B is said to be the cause of A if and only if 1 - B occurs earlier than A, 2 - the conditional probability of A occurring when B occurs is greater than the unconditional probability of A occurring. This definition makes of statistical induction a primary way of structuring knowledge by including causal links in it.

3.8 - Conclusion: induction structures knowledge into several nets

One can say that inductions create a net of implications among pieces of knowledge otherwise independent.
This net of implication must obviously be coherent, a simple requirement that forces the existence of quite strong structures within the net since implication chains pointing at any A are forbidden to lead to ¬A. A well-known example of such chains are the branches of a taxonomy of generality as natural sciences often produce.
We must also stress that these implications have different semantics depending on their origin. For instance, an implication originating from an induction on causality will transfer causal relations, and no property inheritance, while an implication coming from an induction about generality will transfer no causality but property inheritance. In other words, knowledge is represented by a network of implications. These implications are labelled by their semantics, such as IS_A, CAUSES, HAS_PROPERTY, PART_OF, depending which kind of induction each link originates from. For instance, consider the taxonomy of fungi, as given by the natural sciences. We can also consider the functionalities of the fungi, say, their edibility/poisonousness, and the associate networks of causalities. This gives two different networks of implications that must be coherent.

4 - Examples of inductive learning

Let us present here nine induction methods. The seven first ones are stemming from the ML community, the last two ones are born from independent communities. Except structural matching, all of them rely essentially on a zeroth order representation. It is thus a good place to discuss why so many inductive systems are zeroth-order logic, and what can be gained (or lost) by going up to first order.
In theory, first order allows the presence of universally or existentially quantified variables. As an example, think of the description of a chemical molecule. Some molecules may have the property that all carbons are linked to at least one hydrogen. This is a typical property that cannot be described but with a universally quantified variable. Another molecule might be described by that that it contains at least one oxygen molecule. This is a typical property that cannot be described but with an existentially quantified variable.
In practice, we would like to draw here some of the consequences of such representation choices. In the above examples, we have been expressing relations among atoms in order to define properties of molecules. More generally, first order well expresses such relations, while zeroth order is very poor at it. As a simpler example, consider the recognition of a figure that contains at least two segments, perpendicular to each other, and having a

common extremity. This property can well be represented in zeroth order, for instance in a feature-value representation such as: (angle_segment$_1$_segment$_2$ = 90°) & (end$_1$_segment$_1$ = end$_1$_segment$_2$).
Suppose now that we want to use this knowledge to recognize the presence of such segments. Suppose that the figure contains many segments, and that the naming of the segments is such that the perpendicular segments are named segment$_n$, $n \neq 1$ or 2, then, to zeroth order, (angle_segment$_1$_segment$_2$ = 90°) & (end$_1$_segment$_1$ = end$_1$_segment$_2$) is totally different from, say, (angle_segment$_3$_segment$_4$ = 90°) & (end$_1$_segment$_3$ = end$_1$_segment$_4$). Thus zeroth order representation fails recognizing the scene. This problem can be avoided when we dispose of an unambiguous way of naming objects. When this restriction is met, zeroth-order representations are quite efficient.
In first order logic, the property we are expressing can be written as: (angle(x_1, x_2) = 90°) & (end(x_1) = end(x_2)), where it is understood that we mean that there exist such x_1 and x_2. Then the recognition system can look at all pairs of segments, select those that are perpendicular, and then, among them, select those that have a common end.
This example shows a second consequence of first order logic of high practical importance: all combinations of instances that satisfy a given relation are tried out. This feature is desirable since it exhausts the possibilities of the patterns to be recognized. This feature becomes unbearable whenever there are too many possibilities. Then, one has to choose heuristics in order to cut down the complexity.
In the context of learning, this exploration of combination takes also place at the moment a recognition function is invented. For instance, suppose that we want to learn about fathers, and that we have three examples of fathers, John, Peter, and Tom. Suppose that John and Peter have A grade type children, and that John and Tom have sport fanatics' children. If we start generalizing on John and Peter, we lose the knowledge that John's children are also sport fanatics. Generalization to first order techniques must thus include mechanisms that optimize the choices done in the ordering of the examples.
In conclusion, the main practical features of first order logic (as opposed to zeroth order representations) are that 1 - it allows to express relations among the features, 2 - it takes into account the possible combinations of the predicates in order to generate recognition functions.

4.1 - ID3

The aim of this method is as follows. Given a set of descriptors, of examples and of concepts the examples belong to, find the most efficient way to classify all the examples under the concept they belong to, i.e., find the most efficient way to "recognize" the examples. The method relies on information

theory, it measures the amount of information associated with each descriptor, and chooses the most informative one. By applying descriptors in succession, a decision tree is built that will recognize the examples. As opposed to similar numerical techniques, ID3 preserves the understandability of its results by avoiding introducing linear combinations of descriptors.
In figure 1, we ordered the methods by the amount of background knowledge they rely on. ID3 is typically knowledge poor. Nevertheless, all the works done on ID3 from the original version (Quinlan, 1983) to its recent improvements (see for instance (Bratko and Lavrac, 1987)) are centered on understandability rather than on efficiency, which, by the way, gives to these works their deep originality as well as their belonging to AI. Besides, it seems that most improvements in understandability have been followed by efficiency increase as well (Bratko, 1988).
For more details, see Kodratoff (1988, 1989, 1990a, 1992).

4.2 - INDUCE

INDUCE builds a function R that recognizes positive examples and rejects negative ones, i.e., a complete and coherent function. In order to achieve this goal, it starts by building the most general recognition function R_i which recognizes the example e_i and rejects the set of examples $\{e_j\}$.

Let POS be the set of positive examples, and NEG be the set of negative examples, let e_i be the current member of POS. The algorithm to compute R is as follows.
First, as we said, suppose that we have been able to compute R_i, the recognition function that recognizes e_i (and some others of POS), and rejects all instances of NEG. If it happens that R_i recognizes all of POS, we have obtained a complete and consistent recognition function, $R = R_i$. If not, choose an other example e_j, and compute R_j. Then, $R_i \vee R_j$ (where $\vee$ is the logical disjunction) will improve on e_k, it is consistent by construction. The process goes on as long $R_i \vee ... \vee R_n$ is not complete.
For more details , see Kodratoff (1988, 1989, 1990a, 1992).

4.3 - Structural Matching (SM)

This is a first order generalization technique. Two formulas structurally match (they "SMatch") if they are identical except for the constants and the variables that instantiate their predicates.

More formally: Let E1 and E2 be two formulae, E1 SMatches E2 if and only if there exists a formula C and two substitutions s_1, s_2 such that:

1-$s_1 \circ C = E1$ and $s_2 \circ C = E2$.

2-s_1 and s_2 never substitute a variable by a formula or a function.

where ∘ denotes the application of a substitution to a formula.

It must be understood that SM may be difficult up to undecidable. Nevertheless, in most cases, one can use the information coming from the other examples, in order to know how to orient the proofs necessary to the application of this definition. Even if SM fails (which may often occur), the effects of the attempt to put into SM may still be interesting. We say that two formulae have been SMatched when every possible property has been used in order to put them into SM. If the SM is a success, then SMatching is identical to putting into SM. Otherwise, SMatching keeps the best possible result in the direction of matching formulae.
For more details, see Kodratoff (1988, 1989, 1990a, 1992) and Vrain (1990).

4.4 - Knowledge refinement

Most inductive machine learning techniques produce knowledge bases that need to be verified in order to find possible errors. Once errors are found, machine learning techniques can be used again to propose corrections to the knowledge base. In contrast to classical inductive techniques, knowledge base refinement has to be knowledge intensive and incremental.
Systems typical of this approach are CLINT (De Raedt and Bruynooghe, 1989; De Raedt 1991), DISCIPLE (Tecuci and Kodratoff, 1990). and MOBAL (Morik, 1988). All three systems generate examples that are accepted or rejected by the user, and they refine their knowledge in accordance with the user's reactions. DISCIPLE relies on the relevance of explanations for success and failure, proposed to the user, and maybe improved by him. MOBAL and CLINT rely on consistency checking, and both are based on logic programming. They draw the consequences of the acquired knowledge and ask advice to their user when an inconsistency is met. CLINT learns by increasing the possible different syntax for the clauses (e.g., allowing predicates with one more variable as before) and can use meta-level properties in the form of integrity constraints. MOBAL learns by several means, one of them is a progressive organization of the knowledge in hierarchical structures, another one is the propagation of consequences of second order rules, i.e., rules that describe properties of rules.

4.5 - Inductive Logic Programming

This very new sub-domain of ML is concerned by a rationalization of the inductive steps performed when writing a program in a logic programming language. It can be seen as represented by two approaches. The semantical approach (Muggleton and Buntine, 1988; Rouveirol and Puget, 1990) is based on the principle of inverse resolution and its derivations. The

syntactical approach (Shapiro, 1983; Quinlan, 1991) is derived from classical generalization methods and operators.
The main difference between the two directions is that the syntactical approach performs small syntactic generalization or specialization steps (e.g., remove a condition or add one) while the semantical direction takes into account the available knowledge, and allows itself larger inductive steps.

4.6 - Scientific discovery

Several early computer programs have been constructed that simulate various aspects of discovery (Lenat, 1977, 1983; Buchanan, 1978; Langley, 1981; Michalski and Stepp, 1981; Langley, Zytkow, Simon, and Bradshaw, 1983). Although they show interesting performance on various discovery tasks, each has a very narrow scope of applicability, as compared to human capabilities, since the whole process of scientific discovery is too complex. Nevertheless, an approximation to it can be found in the following five steps

- cluster the data according some similarities
- hypothesize the form of the laws that can possibly fit the data
- discover the exact form of law describing some regularity among the data
- evaluate the law by examining how well it fits the data

In the last few years, steps towards a complete computational model of the scientific discovery process, integrating these five steps, have been attempted, at least partially. These systems introduce the ability to determine the conditions of the applicability of a law (the ABACUS systems, Falkenhainer and Michalski, 1990; FAHRENHEIT, Zytkow, 1987; Koehn and Zytkow, 1986; ARC, Moulet, 1991), the ability to design experiments (KEKADA, Kulkarni and Simon, 1988; FAHRENHEIT, Koehn and Zytkow, 1986), the ability to represent processes (Forbus, 1984), the ability to reason by analogy (Falkenhainer and Rajamoney, 1988; Falkenhainer, 1987), and the ability to revise a theory (STAHL, Zytkow and Simon, 1986; STAHLp, Rose and Langley, 1986; Rajamoney, 1988). An integration of all these abilities can be found in Sleeman's informal qualitative models (IQMs) (Sleeman et al., 1989).

4.7 - Conceptual clustering

Conceptual clustering is defined by the tight interaction existing between the clustering and its meaning. Classical Data Analysis is certainly able to compute clusters of examples, and to find generalizations that describe each cluster. What is new in conceptual clustering is that the result of the clustering, i.e., a set of related examples, is performed by using a quality criterion relative to the concept that characterizes the cluster. This demands a kind of utility measure which is more complex than the distances used by Data Analysis.

4.7.1 - COBWEB (Fisher, 1987)

COBWEB is a system that performs conceptual clustering in zeroth-order logic (using an attribute-value representation), and in an incremental way. It uses a measure, named category utility, that heuristically evaluates the utility of a concept, relative to its degree of generality. A useful concept is a tradeoff between very general concepts that are not precise enough, and very particular concepts that are lengthy to retrieve. In practice, Fisher uses a utility measure defined by Gluck and Corter (1985). It is a combination of intra-class similarity (which reflects how much predictable is a value for class members) and of inter-class dissimilarity (which reflects how much a value is predictive for a class). For instance, intra-class similarity for class C_k is computed by the probability for feature F_i to take value V_{ij}. It is written as $P(F_i = V_{ij} \mid C_k)$. Considering the concept of *mammal*, the probability P(habitat = lives_on_ground | mammal) is the ratio of the count of examples that describe a mammal that lives on ground to the total number of examples of mammals. On the contrary, interclass similarity depends on $P(C_k \mid F_i = V_{ij})$ since the larger this probability, the fewer the objects in different classes share value V_{ij}, and the more predictive is the value for the class.
The value of the utility measure is computed from the actual examples. It allows to place a new example in an existing class. Whenever this operation is less interesting in terms of utility measure than creating a new class, a new class is created. It may also happen that merging two classes, or splitting an existing class are more efficient, still in terms of utility measure. COBWEB accordingly reacts to the introduction of a new example.
Thus, based on the utility measure, new concepts are created or destroyed incrementally with the input of new examples. A concept is characterized by an attribute-value probability. In that sense, concepts in COBWEB are probabilistic concepts.
Once the learning is considered as completed, COBWEB has generated a classification tree for examples it has been learning from, and each node of the tree is a concept which has been thus discovered by the system.

4.7.2 - KBG (Knowledge-Based Generalization) (Bisson, 1991)

KBG is a system that performs conceptual clustering in first-order logic, and in a non-incremental way.
As we already stated, the main features of first order logic (as opposed to

zeroth order representations) are that 1 - it allows to express relations among the features, 2 - it takes into account the possible combinations of the predicates in order to generate recognition functions. Thus, in KBG, the utility measure is not linked to a statistical count of the occurrences since the total number of occurrences cannot be too high unless it generates a combinatory explosion. It is rather linked to a similarity measure that takes into account the existence of n-ary relations. For instance, consider the similarity between the relations father(John, Peter) and father(Tom, Betty). It is clear that the similarity between John and Tom depends on many factors, among which the similarity between Peter and Betty. Depending on the context, the function measuring the similarity between Peter and Betty can even induce greater similarity between John and Tom (say, they are both parents of teenagers) or reduce the similarity between John and Tom (say, the problems of a parent of a female teenager may be widely different of those of the parent of a male teenager). Inversely, the similarity between Peter and Betty depends also of the similarity between John and Tom (say, they are both children of middle-class parents).The similarity measure in KBG takes into account this transmission of similarities among siblings of the same predicate.
Once the similarity among all objects in computed, they are clustered according to this measure, depending on user-fixed thresholds to define the distance in similarity authorized inside a class. The most central exemplar of a class is chosen as prototype and generalization is then driven by the prototype.

4.8 - Genetic algorithms

They can be viewed as a new search technique, original by the two following features.

- The search is done in parallel by numerous individuals,
- some individuals are very far from the mean, in order to check regions that are far from the one where convergence is currently taking place.

Choice of a representation (and an example problem)

Consider a robot moving in an environemnt with obstacles, as schematized in figure 2.
The detection of an obstacle will be represented by a string of bits in which the nth bit is at 0 if there is no obstacle in the nth direction, at 1 otherwise.
In figure 2, the robot sees obstacles in directions 2, 5 and 6. This is represented by giving the value 1 to bits 2nd, 5th and 6th bits of a vector of bits: (01001100).

The direction of motor 1 is the same as the direction of detector 1. We represent that a motor is active by giving the value 1 to the corresponding bit.

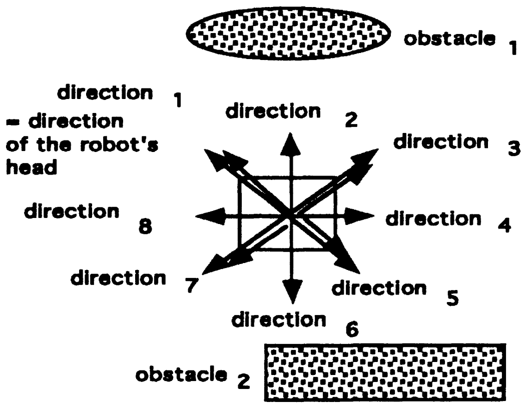


Figure 2. The central square represent the robot. It can detect obstacles in directions 1 to 8, but it can move only in direction 1, 3, 5, and 7.

Thus, for instance, that only motor 1 is active is represented by (1000), that only motor 2 and motor 4 are active is represented by (0101).
The chains of bits can also contain the symbol "*", meaning that the value of the corresponding bit is either 0 or 1. Therefore, this symbol plays the role of a variable for the GAs.
The command of the robot is thus represented by a set of rules of the form (********) $\rightarrow$ (****). The left part of the rule corresponds to the environment of the robot, and its right part describes the move associated to that environment.

Rewarding and selecting a rule

The so-called "strength" of a rule is defined as its utility in a given environment. This is a function written by the GA's maker. It relies heavily on the maker's knowledge of the application field. A bad definition of the strength of a rule will lead to very slow convergence of the GA.
In our example, let us call F the mean distance of the robot from the p nearest obstacles. Let us then call F_i^+ the value of function F after application of rule i, and F_i^- its value before the application of rule i.
Let us call V_i a function of the robot's speed after application of rule i. V_i is 0 if the robot is not headed towards an obstacle and has the value v of the robot's speed if it is headed towards an obstacle.
This will allow us to define a reward function, r_i, for the application of a rule i by: $r_i = F_i^- - F_i^+ - V_i$.
Let then r_i (n) be a function which gives the value of the reward of the rule r_i at its nth application. The strength of a rule varies depending on the number of times it is applied. It is a function $STRENGTH_i(n)$ defined by the mean of all its rewards: $STRENGTH_i(n) = [\Sigma_n r_i(n)]/n$.

Modification of the system of rules

Among K rules, let us choose the M strongest ones. From these M rules, generate P new rules. After a cycle, the system will contain the M strongest rules and their descendent. This selection-reproduction cycle is iterated many times.
There are three main genetic operators used to generate new rules.

- *Duplication*
- *Mutation*

This is a set of admissible changes to be performed on the rules.
For instance, considering the rule (01001100) → (1000), one can become aware that the move is always right, without dependence on the existence of obstacles in 5 and 6. Thus, the following more general rule can be generated: (0100**00) → (1000).

- *Cross-over*

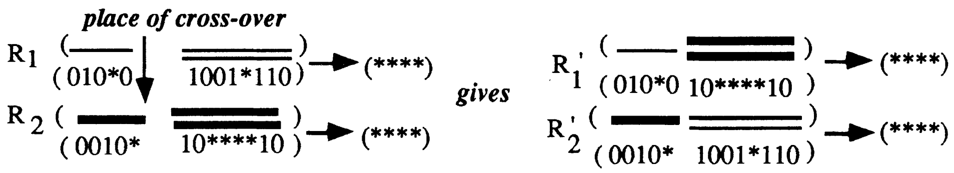

place of cross-over
R1 (010*0 1001*110) → (****)
R2 (0010* 10****10) → (****)
gives
R'1 (010*0 10****10) → (****)
R'2 (0010* 1001*110) → (****)

Figure 3. A graphical example of a cross-over.

The cross-over of (010*0 1001*110) and (0010* 10****10) between the 5th and the 6th bits creates two new chains: (010*0 10****10) et (0010* 1001*110).

4.9 - Neural networks

Since neural networks (NNs) do learn, but are not born within the ML community, they came as a challenge to this community, which started at once establishing comparisons between neural network methodologies and its own (Fisher and McKusick, 1989; Mooney, Shavlik, Towell, and Grove, 1989; Shavlik and Towell, 1990; Weiss and Kapouleas, 1989; Dietterich, Hild, and Bakiri, 1990; Dietterich and Bakiri, 1991). In a few words, it seems that the general conclusions of these comparisons are fivefold.

1 - NNs learn better on noisy data
2 - NNs learn better when background knowledge is poor
3 - NNs learn very slowly
4 - NNs need a large amount of work before they learn properly
5 - NNs do not deliver understandable rules.

We shall address rather on points 2 and 5, which are related with interactions with their field users.
NNs use hidden layers are those between the symbolic attributes describing the concept, and the concept itself. In general, no special semantic is given to

each node in the hidden unit. There are two possible approaches in order to "symbolize" these hidden units. One is to provide them with a meaning, that is, to build NNs with semantically significant hidden units. The other one is to extract rules out of the hidden units.

Building NNs with no "hidden" layers (Shavlik and Towell, 1990; Ragavan and Piramuthu, 1991)

The NN has several layers, intermediate between the symbolic attributes and the final concept, but the structure of the layers and the meaning of each node is well-linked to background knowledge. These systems make use of a tree-building mechanism that allows to give a layer form to the knowledge.
In order to understand how this can be done, let us study the following *ad hoc* example. Suppose that our background knowledge can be summarized by the following five rules, and suppose that the goal of the NN to build is to recognize concept A.

A :- B, E	B :- C, D, I	C:- F
D:- G,H	E:- I,J	

Thus we ask the question A? to our background knowledge and we obtain a proof tree for A, as shown in figure 4a, below.

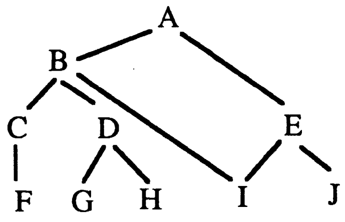


Figure 4a. Proof tree for A.

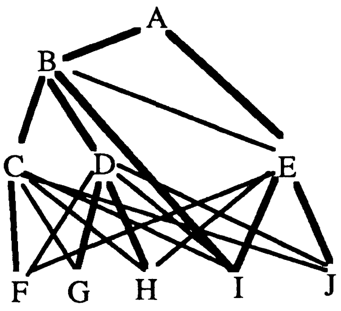


Figure 4b. Structure of the associated NN.

Once this tree has been obtained, it will be used as a starting point for the structure of a NN. For instance, the overall structure of the tree can be kept unchanged, thus the number of hidden units is the tree depth minus two. The existing links are kept and affected with a coefficient one at the start. Within each level, all other possible links are also established, with a coefficient zero. Thus, no new unit can be invented, nor a new link between two non-adjacent levels. Figure 4b shows such a neural structure, where the thick lines are links existing in the proof structure, and the thin lines are the new links introduced with coefficient zero. Back-propagation is then run on this structure and, depending on the examples, the coefficients of old links can be decreased, and the coefficients of new links be increased.

In the final network all links are between nodes loaded with semantics, thus easy to interpret in human terms.
This approach is the one of Shavlik and Towell.
Ragavan and Piramuthu use a very similar technique, except that the initial structure of the NN is obtained by using ID3 to build a decision tree, as explained in section 4.1. Instead of making use of theoretical knowledge available about the field of application, they use the same examples from which the NN is going to learn, but in a symbolic way, in order to have indications on the structure of the best NN.

Extracting rules from NNs with hidden layers (Gallant, 1988; Saito and Nakano, 1988; Fu, 1991)

This task is obviously easy when there are no hidden units. In that case, the final decision is directly linked to the elements of which it depends.
When there are hidden units, the main problem is the one of the large combinatory explosion that takes place when all possible paths are retrieved in order to estimate the contribution of each input to the final output. The number of rules generated will be enormous since each layer influences all other layers. A solution is to introduce heuristics that select more important hidden units, and cut into the number of choices to influence the chosen units. In (Fu, 1991), for instance, a threshold selects which units to consider, and the combinatorics are reduced by considering only the influence of one layer on the following, ordered from the inputs to the outputs of the NN.

5 - Learning by analogy

The analogy scheme we shall describe here is quite a classical one (Winston, 1982; Gentner, 1983; Chouraqui, 1985; Falkenhainer, Forbus, and Gentner, 1986; Carbonell, 1983, 1986; Kedar-Cabelli, 1988; Kodratoff, 1990b). Let us suppose that we dispose of a piece of information, the **base**, that can be put into the form of a doublet (A, B) in which it is known that B depends on A. This dependency will often be causal, and it does not need to be very formal nor strict. In the following, we shall call this relation β, and refer to it as the **causality** of the analogy. The reason for this slip of language from dependency to causality will become clear in the following sections. Suppose now that we find an other piece of information, the **target**, (A', B') that can be put into the same form, and such that there exists some resemblance (similarity) between A and A'. In the following, we shall call this relation α, and refer to it as the **similarity** of the analogy. Let us call β' the causal dependency between A' and B', and α' the similarity between B and B', as shown in the figure below.

$$\begin{tikzcd}
\textbf{BASE} & \text{resemblance/difference relations} & \textbf{TARGET} \\
A \arrow[rr, "\textbf{(SIMILARITY)}", "\alpha"'] \arrow[d, "\beta"'] & & A' \arrow[d, "\beta'", "\text{dependence relations } \textbf{(CAUSALITY)}"'] \\
B \arrow[rr, "\alpha'"] & & B'
\end{tikzcd}$$

Figure 5. The general scheme of analogy.

The analogy problem amounts to find a missing part of this diagram, knowing some others of its parts. For instance, as in Winston (1982), proceeds as follows. The similarity α is not given in advance, but it is a partial matching of A and A', and B is actually the result of this partial matching, The resulting causalities β are the all the causalities inside the discovered B, and one assumes that β' = β. In that way, starting from A and A', and from a similarity measure which is actually a process (partial matching), one progressively discovers the causalities.

In the case of recognition and evaluation of existing analogies, or in the famous case of case-based reasoning, there is no need to draw a difference between similarity and causality. In that case, causality is just one more similarity between source and target. On the contrary, causality is central to the generation of new analogies.

As an illustration, consider the following analogy, proposed in (Russell, 1989).
From *nationality*(Louis, France) & *nationality* (Antoinette, France) & *native_language*(Louis, French), Russel (1989) finds by analogy that *native_language*(Antoinette, French). Adding new information about Louis and Antoinette (we assume here that these characters are the royal couple sent to the guillotine during the French revolution, thus taking Antoinette for Marie-Antoinette), like *lives_in*(Louis, France) and *lives_in*(Antoinette, France) will increase the similarity between Louis and Antoinette, therefore increasing the similarity between target and base. Conversely, adding information like *male*(Louis) & *born_in*(Louis, France) and *female*(Antoinette) & *born_in*(Antoinette, Austria) will decrease theur similarity. With this added information, the analogy can be written without causality, as in figure 6.
On the contrary, as in figure 7, one can also consider that some of this information is causal. It will allow us to find back the given analogy when one considers that β_1 = *lives_in*(Louis, France) and β'_1 = *lives_in*(Antoinette, France) as causalities for the fact of being native French speaker, and when

one does not take into account that Antoinette is born in Austria.

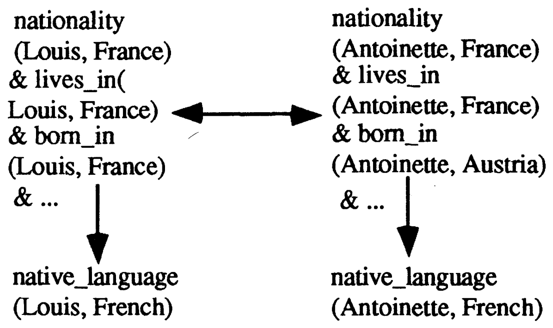

nationality (Louis, France) & lives_in(Louis, France) & born_in (Louis, France) & ...
nationality (Antoinette, France) & lives_in (Antoinette, France) & born_in (Antoinette, Austria) & ...
native_language (Louis, French)
native_language (Antoinette, French)

Figure 6. The given analogy, without causality.

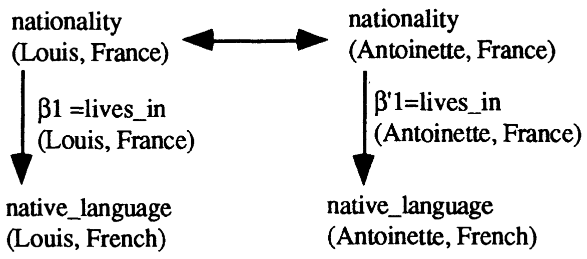

nationality (Louis, France)
nationality (Antoinette, France)
β1 =lives_in (Louis, France)
β'1=lives_in (Antoinette, France)
native_language (Louis, French)
native_language (Antoinette, French)

Figure 7. Inventing again the given analogy by using a causality of the form "x lives_in y" in order to explain that "native_language(x) = language(y)".

Consider now that one adds the following information about Louis: *born_in*(Louis, France). Then, the similar information about Antoinette, β'_2 = *born_in*(Antoinette, Austria), leads to the analogy *native_language* (Antoinette, German), or to *fluent_in* (Antoinette, French), depending on the causality to be used. If β_2 and β'_2 are considered as causal and β_1 and β'_1 are considered as factual, then the analogy should give *native_language*(Antoinette, German).

In this analogy, one is implicitly using theorems of the kind: $\forall$x [*born_in*(x, France) $\Rightarrow$ *native_language*(x, French)] and $\forall$x [*born_in*(x, Austria) $\Rightarrow$ *native_language*(x, German)].

The choice of using these theorems follows from the choice of causality. Let us show why in three steps.

First step: Understanding causality. In the present case, the causality is β_2 = *born_in*(Louis, France), which "explains" why *native_language*(Louis, French).

$$\begin{tikzcd}
\text{nationality (Louis, France) \& lives\_in (Louis, France)} \arrow[r, leftrightarrow] \arrow[d, "\beta 2 = \text{born\_in (Louis, France)}"] & \text{nationality (Antoinette, France) \& lives\_in (Antoinette, France)} \arrow[d, "\beta' 2 = \text{born\_in (Antoinette, Austria)}"] \\
\text{native\_language (Louis, French)} & \text{native\_language (Antoinette, German)}
\end{tikzcd}$$

Figure 8. Inventing another analogy by using a causality of the form "x *born_in* y" in order to explain "*native_language*(x) = *language_of*(y)".

From this, we can infer that the analogy has been using ways of deducing the result from its causality. Therefore, we have to consider theorems that have a generalization of *born_in*(Louis, France) in their premise, and that have a generalization of *native_language*(Louis, French) in their conclusion. In other words, we have to consider the different ways by which one might prove something of the form *native_language*(x, y) from something of the form *born_in*(x, y). This may be very difficult, and the difficulty of finding the link between the causality and its consequences may become a huge task by itself. In the very case we are looking at presently, this inference can be done in a single step by using the theorem ∀x [*born_in*(x, France) ⇒ *native_language*(x, French)].

Second step: Using similarity. Similarity tells us that Louis in the base must be replaced by Antoinette in the target. Therefore, we guess that the causality in the target is β'_2 = *born_in*(Antoinette, Austria).

Third step: Combining causality and similarity. We look for theorems the premise of which is a generalization of *born_in*(Antoinette, Austria), and the conclusion of which is a generalization of *native_language*(x, y). Once more, this step may be very complicated but, in this case, we find in one step that ∀x [*born_in*(x, Austria) ⇒ *native_language*(x, German)] is the looked for theorem. Applying it to the premise *born_in*(Antoinette, Austria) leads to the conclusion *native_language*(Antoinette, German), which becomes the conclusion of our analogy, as shown in figure 8.

By changing the knowledge considered as causal, the result of the analogy will change accordingly. In our example, it is easy to see that considering such as, by choosing a causality of the form "x *lives_in* y", thus calling to theorems such as ∀x [*lives_in*(x, France) & *born_in*(x, France) ⇒ *native_language*(x, French)], or ∀x [*lives_in*(x, France) & ¬*born_in*(x, France) ⇒ *fluent_in*(x, French)], then, the analogy will yield *fluent_in*(Antoinette, French).

When creating analogies, the choice of an information as causality will orient the invention process. When analyzing existing analogies, all kinds of information play a role in rating the given analogy. For instance, in the case of the given analogy above, one might well use both information *lives_in*(Louis, France) and *born_in*(Louis, France) to rate the given analogy. On the contrary, when creating analogies, one has to choose between the available information which one is of causal nature, and this choice changes the output of the analogy process. In other words, from the analysis point of view, *native_language*(Antoinette, German) is as good an analogy as *fluent_in*(Antoinette, French), while from the invention point of view, they differ in the information that has been chosen as causal. In practice, one should always dispose of large amounts of theorems such as those exemplified in the "Antoinette" example, above,, possibly even of theorems that contradict each other. Analogy, which contains for us a choice of causality, allows to chose which to use.

This discussion allows to understand how case-based reasoning (CBR) works since it takes exactly the position of recognizing an analogy as defined above. Very elementary case-based reasoners (as seem to be the present available shells such as CBR-REMIND, CBR-Express, and ESTEEM) use a similarity function that does not take into account deep knowledge, while evolved ones, such as in (Kolodner, Simpson, and Sycara, 1985; Sycara, 1990) take into account deep structural knowledge. In both cases, rather than solving each new problem from first principles or rules, CBR accesses previous appropriate experiences, adapts them, and re-uses them in the current situation.
Advantages of CBR include increased efficiency because plans do not have to be created from scratch, failure avoidance and repair using guidance from similar previous failures, dealing with open worlds and situations not easily formalizable, and preventing brittle system behavior.

6 - Conclusion

This paper gives a somewhat biased idea of what has been happening in machine learning since 10 years because it presents very shortly the main inductive techniques, it hints only at the very important topic of case-based reasoning, and skips explanation-based learning in strong and weak theory domains. Referring to "the Machine Learning Series," below, and to the author's already published review papers, the reader should be able to complete this perspective in a satisfying way. Notice, however, that no

complete and comprehensive review of case-based reasoning seems to be presently available.
As it stands, it presents some of the topics that became popular during the last few years, and that are often still a matter of narrow specialization. We hope that the reader is now convinced that many rich combinations have been happening during the last few years, such as the symbolic/numeric approach to connectionism as described in section 4.7.
The field yielded a first crop in the 8O's with ID3 and AQ that have been tested upon hundreds of applications. The second crop, that of the 85's, contains EBL and conceptual clustering. This paper shows that a third one is coming now, with yet new ideas and systems. Everyone knows that (at least some) learning is the key to the future development for many applications: Here are the tools, they should be tried more often.

Acknowledgments
This work has been partially supported by "PRC-GRECO IA" from CNRS and MRT and by ESPRIT project MLT, P2154. We also thank Luc De Raedt, Katia Sycara, Derek Sleeman, and Jan Zytkow, for the use we made of some of their unpublished material.

References

"The Machine Learning Series:"
Machine Learning: An artificial intelligence approach, Vol. 1, Michalski R. S., Carbonell J. G., Mitchell T. M. (Eds.), Morgan Kaufmann, Los Altos, 1983.
Machine Learning: An artificial intelligence approach, Vol. 2, Michalski R. S., Carbonell J. G., Mitchell T. M. (Eds.), Morgan Kaufmann, Los Altos, 1986.
Machine Learning: An artificial intelligence approach, Vol. 3, Kodratoff, Y., Michalski R. S. (Eds.), Morgan Kaufmann, San Mateo CA, 1990.

Angluin D., Laird P. "Learning from Noisy Examples," *Machine Learning 2*, 343-370, 1988.
Bisson G. "KBG: a Generator of Knowledge Bases," in *Machine Learning: Proceedings of EWSL 91*, Y. Kodratoff (Ed.), Porto, Springer-Verlag 482, pp 137-137. An extended version is published in the Proceedings of the International Conference Symbolic Numeric, Data Analysis and Learning, Paris 17-20 Sept. 1991, pp. 399-415.
Bratko I. (1988) Unpublished set of lectures at European Summer School on Machine Learning, Les Arcs, France.
Bratko I., Lavrac N. (Eds) *Progress in Machine Learning,* Sigma Press, Wilmslow 1987.
Buchanan, B.G., and Mitchell, T.M., "Model-Directed Learning of Production Rules", in Waterman A., Hayes-Roth F. (eds.), Pattern-Directed

Inference Systems, Academic Press, New York, 1978.

Carbonell, J.G. "Learning by Analogy: Formulating and Generalizing Plans from Past Experience" in R.S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach,* Morgan Kaufmann 1983, pp. 137-159.

Carbonell, J.G. "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition", in R.S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach, Volume II*, Morgan Kaufmann 1986, pp. 371-392.

Carbonell J.G., Gill Y. "Learning by Experimentation: The Operator Refinement Method", in *Machine Learning: An artificial intelligence approach*, Vol. 3, Kodratoff Y. & Michalski R. S., (Eds.), Morgan Kaufmann, 1990, pp. 191-213.

Carbonell J.G., Knoblock C.A., Minton S., "PRODIGY: An Integrated Architecture for Planning and Learning," in Architectures for Intelligence, K. VanLehn (Ed.), Erlbaum, Hillsdale NJ, 1990.

Chouraqui E. (1985) Construction of a model for reasoning by analogy. Progress in Artificial Intelligence, Steels L., Campbell J.A. (Ed.), Ellis Horwood, London, p.169-183.

Cox P.T., Pietrzykowski T., "Causes for Events: Their Computation and Applications," *Proceedings of the Eighth International Conference on Automated Deduction*, Oxford, 1986, Lecture Notes in Computer Science n° 230, Springer Verlag, Berlin, pp. 608-621.

De Raedt L., Bruynooghe M. Constructive induction by analogy : a method to learn how to learn", Proceedings of the 4th European Working Session On Learning, pages 189--200. Pitman, 1989.

De Raedt. L. "Interactive Concept-Learning, PhD thesis, Department of Computer Science, Katholieke Universiteit, Leuven, 1991.

Dietterich, G. T., Michalski, R. S. "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods" *Artificial Intelligence Journal 16*, 1981, pp. 257-294.

Dietterich T. C., Hild H., Bakiri G. "A Comparative Study of ID3 and Backpropagation for English Text-to-Speech Mapping," Proc. 7th ICML, pp. 24-31, 1990.

Dietterich T. C., Bakiri G. "Error-Correcting Output Codes: A General Method for Improving Multiclass Inductive Learning Programs," Proc. AAAI-91, pp. 572-577, 1991.

Dupré J. "Probabilistic Causality: A Rejoinder to Ellery Eels," *Philosophy of Science 57*, 1990, 690-698.

Duval B., Kodratoff Y. "A Tool for the Management of Incomplete Theories: Reasoning about explanations" in *Machine Learning, Meta-Reasoning and Logics,* P. Brazdil and K. Konolige (Eds), Kluwer Academic Press pp. 135-158, 1989.

Falkenhainer, B., Forbus, K. D., Gentner, D., "The Structure-Mapping Engine", Report N° UIUCDCS-R-86-1275, DCS, Univ. of Illinois at Urbana-Champaign, May 1986. See also Proc. AAAI-86.

Falkenhainer, B.C., "Scientific Theory Formation Through Analogical Inference," Proc. Fourth International Workshop on Machine Learning, June 22-25, Los Altos, CA, Morgan Kaufmann Publ., 1987, pp. 218-229.

Falkenhainer, B.C., and Rajamoney, S., "The Interdependencies of Theory Formation, Revision, and Experimentation", Proc. Fifth International Conference on Machine Learning, June 12-14, Los Altos, CA, Morgan Kaufmann Publ., 1987, pp. 353-366.

Falkenhainer, B.C., and Michalski, R.S., "Integrating Quantitative and Qualitative Discovery in the ABACUS System," in *Machine Learning: An artificial intelligence approach, Vol. 3,* Kodratoff, Y., Michalski R. S. (Eds.), Morgan Kaufmann, San Mateo CA, 1990, pp. 153-190.

Fisher D. H. "Knowledge Acquisition Via Incremental Conceptual Learning," *Machine Learning 2,* 139-172, 1987.

Fisher D. H., McKusick K. B. "An Empirical Comparison of ID-3 and Back-Propagation", Proc. IJCAI-89, pp. 788-793, 1989

Forbus, K.D., "Qualitative Process Theory", in D.G. Bobrow (ed.), Qualitative Reasoning about Physical Systems, Cambridge, MA: MIT Press, 1984.

Franova M. "Fundamentals for a new methodology for inductive theorem proving: CM-construction of atomic formulae; in: Y. Kodratoff, (ed.): Proceedings of the 8th European Conference on Artificial Intelligence; August 1-5, Pitman, London, UK, 1988, 137-141.

Fu L. "Rule Learning by Searching on Adapted Nets", Proc. AAAI-91, pp. 590-595, 1991.

Gallant S. I. "Connectionist Expert Systems", *Com. ACM 31*, 152-169, 1988.

Gentner, D., "Structure-Mapping: A theoretical Framework for Analogy", *Cognitive Science 7*, pp. 155-170, 1983.

Gluck M.A., Corter J.E. "Information uncertainty, and the utility of categories", Proc. Annual Conf. Cognitive Sci. Soc., pp. 283-287, Lawrence Elbaum, Irvine CA, 1985.

Greiner R., Elkan C. "Measuring and Improving the Effectiveness of Representations", Proc. IJCAI-91, pp. 518-524, 1991.

Guiho G. "Automatic Programming Using Abstract Data Types; in Proceedings of the Eight International Joint Conference on Artificial Intelligence; August, Karlsruhe, 1983, 1-10.

Kedar-Cabelli, S. "Toward a Computational Model of Purpose-directed Analogy", in *Analogica*, Prieditis A. (Ed), Pitman, London, 1988, pp. 89-107.

Kodratoff Y., "Generalizing and Particularizing as the Techniques of Learning," *Computers and Artificial Intelligence* 2, 1983, 417-441.

Kodratoff Y. "An Introduction to Machine Learning; Pitman, London, 1988.

Kodratoff Y., Apprentissage: Science des explications ou science des nombres?, Annales Telecom. 44, 1989, pp. 251-264.

Kodratoff, Y., Chapter 8: Machine Learning, in *Engineering*, Adeli (Ed), pp. 226-255, 1990a.

Kodratoff Y. "Combining Similarity and Causality in Creative Analogy" Proc. ECAI-90, Carlucci Aiello L. (Ed.), Pitman, pp. 398-403, 1990b.

Kodratoff Y., Diday E. (Eds) *Induction symbolique et numérique à partir des données*, CEPADUES édition, Toulouse, 1991.

Kodratoff Y. "Characterising Machine Learning Programs: A

European Compilation", in *Artificial Intelligence, Research Directions in Cognitive Science, European Perspectives, Vol. 5*, D. Sleeman and N. O. Bernsen (Eds), Lawrence Erlbaum, 1992, pp.79-144.

Koehn, B., Zytkow, J.M., "Experimenting and Theorizing in Theory Formation", in Z. Ras (ed.), Proceedings of the International Symposium on Methodologies for Intelligent Systems, Knoxville, TN, 1986, ACM SIGART Press, pp. 296-307.

Kolodner, J.L., Simpson, R.L., Sycara, K. "A Process Model of Case-Based Reasoning in Problem Solving", Proc. IJCAI, pp. 284-290, 1985.

Kulkarni, D., and Simon, H.A., "The Processes of Scientific Discovery: The Strategy of Experimentation", Cognitive Science, Vol. 12, 1988, pp. 139-175.

Laird J. E., Newell A., Rosenbloom P. L. (1986) *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer, Dordrecht.

Laird J. E., Rosenbloom P. L., Newell A. "Soar: An Architecture for General Intelligence", *AI Journal 33*, 1987, 1-64.

Langley, P.W., "Data-driven Discovery of Physical Laws", Cognitive Science, 5, 1981, pp. 31-54.

Langley, P., Zytkow, J.M., Simon, H.A., Bradshaw, G., "The Search for Regularity: Four Aspects of Scientific Discovery", in Machine Learning: An Artificial Intelligence Approach, Vol. 2, Michalski R.S., Carbonell J.G., and Mitchell T.M. (Eds.), Morgan Kaufman, Los Altos CA 1986, pp. 425-470.

Langley, P., Simon, H.A., Bradshaw, G., Zytkow J.M., Scientific Discovery: An Account of the Creative Processes, MIT Press, Boston, MA, 1987.

Lenat, D.B., "Automated Theory Formation in Mathematics," Proc. Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 833-842.

Lenat, D.B., "The Role of Heuristics in Learning by Discovery: Three Case Studies", in Michalski R.S., Carbonell J.G., and Mitchell T.M. (Eds.), Machine Learning: An Artificial Intelligence Approach, Tioga Press, Palo Alto, CA, 1983.

Manna Z., R. Waldinger Z. "A Deductive Approach to Program Synthesis; ACM Transactions on Programming Languages and Systems, Vol. 2., No.1, January, 1980, 90-121.

Michalski R. S. (1983) "A Theory and a Methodology of Inductive Learning", in *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, pp 83-134.

Michalski R. (1984) "Inductive learning as rule-guided transformation of symbolic descriptions A theory and implementation", in *Automatic Program Construction Techniques*, Biermann, Guiho and Kodratoff editors, Macmillan Publishing Company, New York, pp. 517-552.

Michalski, R. S., Kodratoff, Y. "Research in Machine Learning: Recent progress, Classification of Methods, and Future Direction" in *Machine Learning: An Artificial Intelligence Approach, Volume III*, Y. Kodratoff and R.S. Michalski (Eds.), Morgan Kaufmann, San Mateo, 1990, pp. 3-30.

Mooney R., Shavlik J., Towell G., Grove A. "An Experimental

Comparison of Symbolic and Connectionist Learning Algorithms" Proc. IJCAI-89, pp. 775-780. Also in *Readings in Machine Learning*, Shavlik J. W. and Dietterich T. G. (Eds), Morgan-Kaufmann, San Mateo, CA 1990, pp. 171-176.

Morik K. "Acquiring Domain Models", in Knowledge Acquisition Tools for Expert Systems 2, Boose J. and Gaines B. (Eds), Academic Press, 1988.

Moulet M. "Using accuracy in law discovery", Fifth European Working Session on Learning, Porto, mars 1991, Springer Verlag, pp. 118-136.

Muggleton S., Buntine W., "Machine Invention of First-Order Predicates by Inverting Resolution," Proc. 5th Internntl. Conf. on Machine Learning, Morgan Kaufmann CA 1988, pp. 339-351.

Peirce C. S. "Elements of Logic" in *Collected Papers of Charles Sanders Peirce (1839-1914)*, C.H. Hartshone and P. Weiss (Eds), Harvard University Press, Cambridge MA, 1965.

Priedetis A. (1988) (editor) *Analogica* Pitman, London.

Quinlan J.R. (1983) "Learning Efficient Classification Procedures and their Application to Chess End Games" in *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, pp 463-482

Quinlan J.R. "Determinate Literals in Inductive Logic Programming," Proceedings of the twelfth IJCAI, Morgan Kaufmann Publisher, Inc, San Mateo, CA, Volume 2, pages 746 à 750, 1991.

Ragavan H., Piramuthu S. "The Utility measure of Feature Construction for Back-Propagation," Proc. IJCAI-91, pp. 844-848, 1991.

Rajamoney, S.A., "Explanation-Based Theory Revision: An Approach to the Problems of Incomplete and Incorrect Theories," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1988.

Rose, D., Langley, P., "Chemical Discovery as Belief Revision", Machine Learning 1, 1986, pp. 57-95.

Rouveirol C, Puget J.F. "Beyond inversion of resolution,"Proceedings of the 7th International Conference on Machine Learning, Morgan Kaufmann, 1990.

Rouveirol C. "Semantic Model for Induction of First Order Theories," Proceedings of the twelfth IJCAI, Morgan Kaufmann Publisher, Inc, San Mateo, CA, Volume 2, pp. 685 à 690, 1991.

Russell S.J. *The Use of Knowledge in Analogy and Induction*, Pitman, London 1989.

Saito K., Nakano R. "Medical Diagnostic Expert Systems based on PDP Model", Proc. IEEE Conf. on Neural Networks, pp. 255-262, 1988.

Sato M. "Towards a Mathematical Theory of Program Synthesis; Proc. 6th IJCAI, Tokyo, 1979, 757-762.

Schank R. C. *Explanation Patterns: Understanding mechanically and Creatively*, Ablex Publishing Company, (1987).

Schank R. C., Abelson R. P., *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum, Hillsdale, N.J. (1977).

Schank R. C., Kass A. "Explanations, Machine Learning, and Creativity", in *Machine Learning: An artificial intelligence approach*, Vol. 3, Kodratoff Y. & Michalski R. S., (Eds.), Morgan Kaufmann, 1990, pp. 31-48.

Shapiro E., Algorithmic Program Debugging, The MIT press, 1983.

Shavlik J.W., Towell G.G. "An Approach to Combining Explanation-Based and Neural Learning Algorithms" in *Readings in Machine Learning*, Shavlik J. W. and Dietterich T. G. (Eds), Morgan-Kaufmann, San Mateo, CA 1990, pp. 828-840.

Sleeman D. H., Stacey M.K., Edwards P., Gray N.A.B. "An Architecture for Theory-Driven Scientific Discovery", Proc. EWSL 89, Morik K. (Ed.), Pitman 1989, pp. 11-24.

Smith D. R. "Top-Down Synthesis of Simple Divide and Conquer Algorithm; Artificial Intelligence, vol. 27, no. 1, 1985, 43-96.

Stepp R.E, Michalski R.S. (1983) "Learning from observation: Conceptual Clustering" in *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, pp 331-363

Suppes P. "Conflicting Intuitions about Causality," in *Midwest Studies in Philosophy IX, Causation and Causal Theories*, University of Minnesota Press, Minneapolis 1984, pp. 151-168.

Sycara, K. "Persuasive Argumentation in Negotiation", Theory and Decision, Vol. 28, No. 3, pp. 203-242, May 1990.

Tecuci G., Kodratoff Y. "Apprenticeship Learning in Imperfect Domain Theories," in *Machine Learning: An artificial intelligence approach, Vol. 3,* Kodratoff, Y., Michalski R. S. (Eds.), Morgan Kaufmann, San Mateo CA, 1990, pp. 514-552.

Valiant L. G. "A Theory of the Learnable", *Com. ACM 27*, 1134-1142, 1984.

Vrain C. "OGUST: A System that Learns Using Domain Properties Expressed as Theorems", in *Machine Learning, an Artificial Approach,* volume III, Y. Kodratoff, R. Michalski (Eds), Morgan Kaufmann Publishers, p. 360-382, 1990.

Weiss S.M., Kapouleas I. "An Empirical Comparison of Pattern Recognition, Neural Nets, and Machine Learning Classification Methods" Proc. IJCAI-89, pp. 781-787. Also in *Readings in Machine Learning*, Shavlik J. W. and Dietterich T. G. (Eds), Morgan-Kaufmann, San Mateo, CA 1990, pp. 177-183.

Winston P. H. (1982) "Learning New Principles from Precedents and Exercises", *AI Journal 19*, 321-350.

Zytkow, J.M., Simon, H.A., "A Theory of Historical Discovery: The Construction of Componential Models," Machine Learning, 1, 1986, pp. 107-36.

Zytkow, J.M., "Combining Many Searches in the FAHRENHEIT Discovery System", Proc. 4th International Workshop on Machine Learning, Los Altos, CA, June 1987, Morgan Kaufmann Publ., pp. 281-287.

Zytkow, J.M., Erickson, M., "Tactical Manager in a Simulated Environment", in Ras Z. and Zemankova, M. (eds.), Methodologies for Intelligent Systems, Elsevier Science Publ., 1987, pp. 139-147.

Zytkow J.M., "Deriving Basic Laws by Analysis of Processes and Equations," in Langley, P. and Shrager J. (eds.), Computational Models of Scientific Discovery, 1989.

DESIGNING LOGIC PROGRAMMING LANGUAGES

J.W. LLOYD
Department of Computer Science,
University of Bristol,
University Walk, Bristol, BS8 1TR, U.K.

Abstract. The design of logic programming languages is discussed with emphasis on two aspects: improving the declarative semantics and improving the software engineering facilities of current languages. These discussions are illustrated with example programs in the language Gödel, which is under development at Bristol University. The main design aim of Gödel is to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog. The paper concludes with some remarks about future directions for the design of logic programming languages.

1. Design Issues for Logic Programming Languages

In this section, we discuss various issues for the design of logic programming languages. We begin this discussion by giving the underlying ideas of logic programming and declarative programming.

The starting point for the programming process is the particular problem that the programmer is trying to solve. The problem is then formalised using a language which we assume here to be a (typed) first order language (see (Llo87)) including a set of constants, functions, propositions and predicates. More precisely, the problem is formalised as an interpretation (called the *intended* interpretation) of this language. The intended interpretation specifies the domain of the problem, the meaning of the constants and functions in this domain, and the meaning of the propositions and predicates. In practice, the intended interpretation is rarely written down precisely, although in principle this should always be possible.

Now, of course, it is taken for granted here that it *is* possible to capture the intended application by a first order interpretation. Not all applications can be (directly) modelled this way and for such applications other languages and formalisms may have to be employed. However, a very large class of applications can be modelled naturally by means of a first order interpretation. In fact, this class is larger than is sometimes appreciated. For example, it might be thought that such an approach cannot (directly) model situations where a knowledge base is changing over time. Now it is true that the intended interpretation of the knowledge base is changing. However, the knowledge base should properly be regarded as data to various meta-programs, such as query processors or assimilators. Thus the knowledge base is represented as a ground term at the meta-level, and can be accessed and changed by the assimilator, for example. The meta-programs have fixed intended interpretations which fits well with the setting for declarative programming given above.

P. Dewilde and J. Vandewalle (eds.), Computer Systems and Software Engineering, 263–285.
© 1992 *Kluwer Academic Publishers.*

Based on the intended interpretation, the *logic component* of a program is then written. This logic component is a particular kind of (typed) first order theory which is suitably restricted so as to admit an efficient theorem proving procedure. Commonly, this theory is the completion (Llo87) of a collection of program statements. It is crucial that the intended interpretation be a model for the completion of the logic component of the program. This is because computed answers, which are usually computed by SLDNF-resolution and are known to be correct by the soundness of SLDNF-resolution (Llo87), therefore give instantiated goals which are true in the intended interpretation. Ultimately, the programmer is interested in computing truth in the intended interpretation.

Having written a correct logic component of a program, the programmer then turns to the *control component.* Usually, the computation rule is partly specified by declarations and the pruning of the search tree is specified by a pruning operator, such as commit or cut. To some extent, it is possible to relieve the programmer of responsibility for the control aspects by the use of preprocessors which automatically generate appropriate control. Note that, in practice, control considerations may cause a programmer to go back and rewrite the logic component to improve the efficiency of a program. In any case, a desirable property of the control declarations of a program is that they can be stripped away and what remains is a correct logic component of the program.

The above discussion can be summarised by stating that the central thesis of logic programming is that

- a program is a (first order) theory, and
- computation is deduction from the theory.

It is crucially important that programs can be understood directly as theories. When this is the case, they have simple declarative semantics and can be much more easily verified, transformed, debugged, and so on. Note that because logic programs have a declarative semantics, logic programming is a form of declarative programming and shares all the advantages that declarative programming provides.

As well as having a declarative semantics, logic programs must also have available some basic software engineering facilities. The two most important such facilities are a type system and a module system. We discuss each of these in turn.

The reasons for having types in logic programming languages are well known. The major reason is for knowledge representation. Intended interpretations in most logic programming applications are typed and hence using a typed language is the most direct way of capturing the relevant knowledge in the application. Also, the extra information given by the language declarations of a type system can be used by a compiler to produce more efficient code. Furthermore, type declarations can help to catch programming errors. For example, in an untyped language, simple typographical errors often lead to bizarre program behaviour which can usually only be identified by laborious tracing of the program. In contrast, in a typed language, such errors often lead to syntax errors which can be caught by the compiler. It is nearly always easier to correct an error caught by the compiler than it is to discover and correct an error that leads to wrong program behaviour. Our experience with the Gödel type system (see below) supports the contention that it greatly decreases the effort required for program development and greatly increases the likelihood of correctness of programs compared with untyped languages.

There are two kinds of type systems in logic. The first comes from so-called type theory which applies to higher order logics. This is the foundation for the type systems of functional languages. The other kind of type system studied in logic is called many-sorted logic (End72), (Llo87) and is relevant mainly for first order logic. It is many-sorted logic which provides the foundation for type facilities in (first order) logic programming languages. In the following, to conform to the usual terminology of programming languages, we often refer to a sort as a type.

Many-sorted logic generalises ordinary unsorted first order logic in that it has sort declarations for the variables, constants, functions, and predicates in the language. We can think of ordinary unsorted logic as a special case in which there is only one sort. In the general case, there are a number of (possibly countably infinitely many) sorts. Each constant, for example, is then specified as having a particular sort. There is also a natural definition of what it means for an expression to be a well-formed formula in the many-sorted language. Proof procedures for many-sorted logic can then be defined by a straightforward generalisation of those for unsorted logic. For the declarative semantics, we have to generalise the usual notions of interpretation, logical consequence, and so on.

However, a many-sorted logic alone is not sufficiently flexible for a type system in a logic programming language. The reason is that we want to write predicates which take a variety of types of arguments. For example, the usual *append* predicate will normally be required to append lists each of whose elements has the same fixed but arbitrary type. For this reason, logic programming languages should allow a form of polymorphism, called *parametric* polymorphism, familiar from functional programming languages. Parametric polymorphism necessitates the introduction of type variables, which range over all types. By this means, a polymorphic version of *append*, for example, can be written in a similar way to the functional languages.

Type systems can be based on even richer logics as well. For example, it is common to use an order-sorted logic instead of a many-sorted one. Roughly speaking, an order-sorted logic allows the various domains to overlap instead of being disjoint as for a many-sorted logic. Order-sorted logic has greater expressive power than many-sorted logic, but can have extra implementation difficulties compared with many-sorted logic.

Next we briefly discuss module systems. The usual software engineering advantages of a module system are well known. In its most basic form, a module system simply provides a way of writing large programs so that various pieces of the program don't interfere with one another because of name clashes and also provides a way of hiding implementation details. A satisfactory module system can be based on these standard ideas and thus provide facilities for importation, exportation, and data-hiding. More powerful module systems can be based on providing an algebra of operations on modules, for example, taking the "union" or "intersection" of two modules. Such module systems seem mostly to be at the research level at the moment, although they can be expected to be important in practice in the future, especially when software reuse is more common.

Now to what extent does Prolog, which is by far the most commonly used logic programming language, satisfy the above requirements? Prolog has proved to be a great success in a wide variety of application areas. This success is undoubtedly due to the fact that Prolog is high level, expressive, efficient, and practical. Prolog's importance and widespread use is well justified by these properties. However, Prolog's semantics (and by Prolog, we mean the practical programming language as it is embodied in currently available Prolog systems, not the idealised pure subsets studied in (Llo87), for example) is much less satisfactory. The problems with the semantics are numerous and well known: the lack of occur check, the use of unsafe negation, the use of non-logical predicates, such as *var*, *nonvar*, *assert*, and *retract*, the undisciplined use of cut, and so on. These so-called "impure" aspects of Prolog cause many practical Prolog programs to have no declarative semantics at all and to have unnecessarily complicated procedural semantics. This means that the (systematic) construction, verification, transformation, optimisation, and debugging of many Prolog programs is extremely difficult. Furthermore, Prolog does not have a type system and many Prolog systems have no module system or, at best, a primitive and unsatisfactory one. The conclusion is clear: Prolog is very far from achieving the ideals for a logic programming language and much work on the design of logic programming languages needs to be done.

2. Facilities of Logic Programming Languages

Various facilities which might be provided by a logic programming language include the following:

- types
- modules
- control declarations
- pruning
- meta-programming
- constraint solving
- parallelism
- higher-order
- object-oriented
- functional
- large knowledge bases

We now discuss each of these facilities in turn. (We make no attempt to give complete references for each of these facilities. Instead, we refer the reader to proceedings of recent International and North American Conferences on Logic Programming, which contain many relevant papers.)

We have already discussed types and modules, so we now turn to control declarations. Prolog has a strict left to right computation rule. However, it is now widely appreciated that this rule is too rigid. To express more flexible computation rules, many Prolog systems provide control declarations. One of the most influential systems has been NU-Prolog (TZ88), which provides *when* declarations. These delay a call to a procedure until certain conditions are satisfied, typically that one or more arguments are non-variables or are ground.

It may seem odd to regard pruning as desirable facility of a logic programming language because pruning clearly compromises the declarative semantics of a program. However, the conclusion of (HLS90) is that pruning is useful and acceptable provided the pruning operator(s) provided has certain properties. Prolog uses the cut as its pruning operator. However, cut has a number of semantic problems, which were discussed in detail in (HLS90). The first of these problems is that cut, at least as it is employed in existing Prolog systems, allows considerable uncertainty about what the underlying logic component of the program is. This is because programmers can exploit the sequential nature of cut to leave "tests" out and when this is done the logic component of the program cannot be obtained by simply removing all the cuts from the program. Furthermore, there is no convention for systematically putting back the omitted tests so as to define the logic component precisely. The second problem with cut is that its use, in the presence of negation, can be unsound, in the sense that a computed answer may not be correct with respect to the completion of the logic component of the program. The third problem is that the class of programs containing cut is not closed under program transformations. For these reasons and because the commit of the concurrent logic programming languages (Sha89) has better semantics, it seems preferable to base pruning on the commit of the concurrent languages.

Next we turn to meta-programming facilities. The essential characteristic of meta-programming is that a meta-program is a program which uses another program (the object program) as data. Meta-programming techniques underlie many of the applications of logic programming. For example, knowledge base systems consist of a number of knowledge bases (the object programs), which are manipulated by interpreters and assimilators (the meta-programs). Other important kinds of software, such as debuggers, compilers, and program transformers, are meta-programs.

However, in spite of the fact that meta-programming techniques are widely used, the meta-programming facilities provided by most currently available Prolog and other logic programming systems are by no means satisfactory. For example, important representation (that is, naming) and semantic issues are normally glossed over, since most currently available Prolog systems do not make a clear distinction between the object level and meta-level, and do not provide explicit language facilities for representation of object level expressions at the meta-level. A consequence of the fact that Prolog doesn't handle the representation requirements properly is that it is not possible to (directly) understand most Prolog meta-programs as theories and hence they do not have a declarative semantics. The most obvious symptom of this is with *var*, which has no declarative semantics at all in Prolog. Within the framework of the appropriate representation (HL89), a meta-program is a (typed) first order theory and the meta-logical predicates of Prolog, such as *var*, *nonvar*, and so on, have declarative counterparts.

Meta-programs are often required to manipulate the representations of object programs, creating new object program representations dynamically. For example, a partial evaluator and a program transformer both do this. To do this declaratively, we need one further idea, which is that object programs should be represented not as meta-programs, but instead as meta-level terms (BK82). Using this idea together with the ground representation, it is straightforward to give appropriate definitions for declarative counterparts of the dynamic meta-logical predicates of Prolog, such as *assert* and *retract* (HL88).

There are two basic representations for meta-programming, the non-ground and the ground representation (HL89). The main difference between them is in the way they handle the representation of object level variables. In the non-ground representation, an object level variable is represented by a meta-level variable and in the ground representation by a ground meta-level term. This seemingly small difference has significant consequences. Indeed, it turns out that the ground representation is the one that is needed to provide declarative replacements for Prolog's *var*, *nonvar*, and so on, and that the non-ground representation cannot achieve this at all. Thus the ground representation is easily the more important of the two representations. In fact, the non-ground representation seems useful mostly for the vanilla (Prolog-in-Prolog) interpreter and various extensions of it. As soon as, for example, we want to do any significant manipulation of meta-level terms representing object level expressions, we must use the ground representation. This means that most of the important uses of meta-programming, such as compiling, transforming, partially evaluating, and debugging, will require the ground representation.

Constraint logic programming has become very important over the last couple of years, for the reason that it provides a simple way of solving many problems in operations research, for example, for which the only approach in the past was to write large and complex Fortran (or similar) programs. Instead, constraint logic programming gives programmers the ability to solve substantial constraint problems with comparatively little programming effort. The reason is that much of the sophisticated control needed to solve a constraint problem is built into the constraint language and, therefore, to a large extent programmers only have to write the logic of the problem and can leave the control to the system. The domains covered by constraint systems include the integers, rationals, and the booleans. Constraint logic programming languages currently available include CHIP (DvHS*88), Prolog III (Col90), and CLP(R) (JL87).

Parallelism is naturally a major issue in language design. The approaches to this can be roughly classified as to whether the parallelism is implicit or explicit. In the implicit approach, the parallelism is not explicitly present at all at the language level and all the effort goes into running programs in an efficient a way as possible on a parallel machine by exploiting AND- or OR-parallelism, or both. For example, much of the work on building parallel implementations of Prolog uses this approach. (See, for example, (SWY91a) and (SWY91b).) In the other approach, the parallelism is explicit at the language level. For example, concurrent logic languages (Sha89) have annotations and/or rules defining synchronization and communication, which a programmer can exploit for particular purposes. These languages are not extensions of Prolog since they give up some of Prolog's facilities, such as backtracking and negation. However, they do provide facilities that Prolog does not have, such as the ability to control communication and synchronisation between concurrent processes. The concurrent languages have achieved considerable prominence over the last five years. For example, a concurrent logic programming language, called KL1, is the system language for the ICOT parallel inference machine.

The two families of languages (the Prolog-like ones and the concurrent ones) have developed somewhat independently during the 1980's. However, there now seems the strong chance that the two families can be unified by means of the Andorra principle. This principle is that deterministic goals (that is, those for which at most one clause matches) should be selected first and should be run in AND-parallel mode. Only when there are no deterministic goals is a non-deterministic goal chosen and a choice point created. Many researchers believe that it is possible to design logic programming languages which encompass both the Prolog-like languages and the concurrent languages. (See, for example, (SWY91a), (SWY91b), (HB88), and (Nai88).) This prospect is indeed an attractive one.

Another interesting research area is that of higher-order facilities, as in the language λ-Prolog (NM88). Such facilities provide great expressive power for certain applications, although their implementation can be problematical (for example, higher-order unification is undecidable). While there have been a number of important experiments in higher-order logic programming languages, research still appears to be at the stage of trying to determine just exactly which higher-order logics are the most useful for this purpose and exactly which language features should be provided.

A major trend in programming languages, not just logic programming languages, in recent years has been the introduction of object-oriented features. In many respects, a logic programming language naturally provides many of the features one would expect of an object-oriented language. However, there seems to be no consensus on what facilities an object-oriented logic programming language should actually provide. Furthermore, while there have been a number of attempts at designing object-oriented logic programming languages, none have achieved widespread use so far.

There have also been numerous attempts to combine logic and functional programming. Combining the two paradigms has obvious attractions, but there seem to be many ways to do this and so far there does not appear to be any consensus about which way is best.

The final facility we discuss here is concerned with the handling of large knowledge bases. Logic programming languages naturally provide the kernel of a knowledge base system, since they make available a query processor and, if properly designed, have the necessary meta-logical facilities for handling knowledge bases. However, special difficulties arise if the knowledge bases become large. In this case, the system requires indexing facilities for efficient accessing of knowledge bases and query optimization techniques for efficient answering of queries. Furthermore, if the knowledge base system is to be used concurrently by many users, then database concurrency control techniques need to be implemented. It is likely that other facilities, such as rollback and recovery from errors, will also be needed. There are a number of experimental deductive database systems around the world, but so far these have had very little impact on the commercial database area.

Logic programming language design is currently an area of intense interest and activity, and this section has provided a brief overview of this. In the next few sections, we give an introduction to one particular language currently being developed. In the second last section, the discussion will return to more general issues and some remarks made about future directions for this research.

3. Gödel

The main design aim of Gödel (HL91) is to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog. Gödel is intended to have the same relation to Prolog as Pascal does to Fortran. Fortran was one of the earliest high-level languages and suffered from a lack of understanding at that time of what were good programming language features. In particular, it had a meagre set of data types and relied on the goto for control. Pascal, which was introduced 10 years later, benefitted greatly from a considerable amount of research into programming language design. In particular, Pascal had a rich set of data types and relied on structured control facilities instead of the goto. Similarly, Prolog was designed at the birth of logic programming before researchers had a clear understanding of how best to handle many facilities. Consequently, these facilities compromised its declarative semantics. In the period since Prolog first appeared, various research projects have shown how to design logic programming languages with better software engineering facilities and greatly improved declarative semantics with all the well-known advantages that these bring. The aim was to exploit this research in the design of Gödel.

The declarative nature of Gödel programs has a number of important practical consequences. One is that Gödel more easily allows a parallel implementation. Gödel programs can also be debugged by the very attractive declarative debugging techniques, which are not useful, in practice, for Prolog programs. More formal methods of proving program correctness are also greatly facilitated in declarative programming languages such as Gödel. Finally, the fact that Gödel meta-programs are declarative makes some desirable applications possible. One of these is to build a WAM-like compiler for Gödel by partially evaluating a partial evaluator with respect to an interpreter (which uses WAM-like data structures during interpretation) as input. At the next level, one can obtain a compiler-generator by partial evaluating a partial evaluator with respect to itself as input. Both these applications rely crucially on being able to build self-applicable partial evaluators. No effective self-applicable partial evaluator written in Prolog has ever been built and the prospects for building such a program seem very slim indeed. However, since a partial evaluator in Gödel is declarative, carrying out the same task in Gödel, while difficult, seems eminently achievable. We expect the benefits of the declarative nature of Gödel to become evident partly through the appearance of self-applicable partial evaluators and similarly sophisticated programs over the next couple of years.

The main facilities provided by Gödel are as follows:

- types
- meta-programming
- control
  - control declarations
  - constraint solving
  - pruning operator
- modules
- input/output

We now discuss in some detail the type, module, and meta-programming facilities of Gödel.

4. The Gödel Type System

The Gödel type system is based on many-sorted logic with parametric polymorphism. We first discuss the many-sorted aspect ((End72), (Llo87)) of the type system. Consider module `M1` below which defines the predicates `Append` and `Append3` for appending lists of days of the week. Note that variables are denoted by identifiers beginning with a lower case letter and constants by identifiers beginning with an upper case letter.

Language declarations begin with one of the keywords `BASE`, `CONSTRUCTOR`, `CONSTANT`, `FUNCTION`, `PROPOSITION`, or `PREDICATE`. These declarations declare the *symbols* of the language, which belong in one of the *categories*: bases, constructors, constants, functions, propositions, and predicates. In module `M1`, the language declaration beginning with the keyword `BASE` gives the types of the many-sorted language of the module. It declares `Day` and `ListOfDay` to be *bases*, which are the only types of the language. (More complicated types will be introduced shortly.)

```
MODULE       M1.

BASE         Day, ListOfDay.

CONSTANT     Nil : ListOfDay;
             Monday, Tuesday, Wednesday, Thursday, Friday,
             Saturday, Sunday : Day.
FUNCTION     Cons : Day * ListOfDay -> ListOfDay.
PREDICATE    Append : ListOfDay * ListOfDay * ListOfDay;
             Append3 : ListOfDay * ListOfDay * ListOfDay *
                       ListOfDay.

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
             Append(x,y,z).

Append3(x,y,z,u) <-
             Append(x,y,w) &
             Append(w,z,u).
```

The next three declarations declare the constants, functions, and predicates of the language. The first part of the CONSTANT declaration declares Nil to be a constant of type ListOfDay. The second part declares Monday, Tuesday, etc., to be constants of type Day. The FUNCTION declaration declares Cons to be a binary function which maps a tuple of arguments, where the first argument is of type Day and the second argument is of type ListOfDay, to an element of type ListOfDay. The PREDICATE declaration declares Append to be a ternary predicate each of whose arguments has type ListOfDay. It also declares Append3 to be a quaternary predicate each of whose arguments has type ListOfDay. Statements and goals are written in the language defined by the language declarations.

Module M1 forms a complete Gödel program on its own. A typical goal for this program is as follows.

```
<- Append3(x,y,z,Cons(Monday, Cons(Tuesday,Cons(Wednesday,Nil)))).
```

Next we introduce *constructors* using module M2, which is a variation of module M1. The main difference between the two modules is that in module M2 a unary constructor List has been declared. From the base Day and the constructor List, the set of all types of the language is obtained by forming all "ground terms" from the "constant" Day and the "function" List. Thus the types of the language are Day, List(Day), List(List(Day)), ...

```
MODULE        M2.

BASE          Day.
CONSTRUCTOR   List/1.

CONSTANT      Nil : List(Day);
              Monday, Tuesday, Wednesday, Thursday, Friday,
              Saturday, Sunday : Day.
FUNCTION      Cons : Day * List(Day) -> List(Day).
PREDICATE     Append : List(Day) * List(Day) * List(Day);
              Append3 : List(Day) * List(Day) * List(Day) *
                        List(Day).

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
              Append(x,y,z).

Append3(x,y,z,u) <-
              Append(x,y,w) &
              Append(w,z,u).
```

All bases and constructors[1] of the language must be declared. If there is no constructor, then the set of all types is just the set of all bases. If at least one constructor is declared, then the set of all types is obtained by applying the above construction using the bases as "constants" and the constructors as "functions" of the appropriate arity. For example, if a base `People` were declared in addition to those bases declared in module `M2`, then the set of all types would be `Day`, `People`, `List(Day)`, `List(People)`, `List(List(Day))`, `List(List(People))`, ...

Next we introduce the second aspect of the type system, which is parametric polymorphism. It is common for a programmer to want to write a definition of a predicate for which the arguments of the predicate can have a variety of types. For example, the `Append` predicate is normally written so that it can append lists of any type. For this purpose, we add parametric polymorphism to the type system, as illustrated by module `M3`.

The logic on which module `M3` is based is called *polymorphic many-sorted logic*. In module `M3`, `alpha` is a *parameter*, which is a type variable. A parameter can be instantiated to any type of the language of the logic. Like variables, parameters are not declared. For a polymorphic many-sorted language, we need to extend

[1] Note that a constructor itself is not a type.

```
MODULE       M3.

BASE         Day, People.
CONSTRUCTOR  List/1.

CONSTANT     Nil : List(alpha);
             Monday, Tuesday, Wednesday, Thursday, Friday,
             Saturday, Sunday : Day;
             Fred, Bill, Mary : People.
FUNCTION     Cons : alpha * List(alpha) -> List(alpha).
PREDICATE    Append : List(alpha) * List(alpha) * List(alpha);
             Append3 : List(alpha) * List(alpha) * List(alpha) *
                       List(alpha).

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
             Append(x,y,z).

Append3(x,y,z,u) <-
             Append(x,y,w) &
             Append(w,z,u).
```

the concept of a type. For such a language, a type is a "term" constructed using the bases as "constants", the parameters as "variables", and the constructors as "functions". Thus, for module M3, the following are types: List(alpha), alpha, and List(List(Day)). A *ground* type (also called a *monotype*) is a type not containing parameters. Thus the set of ground types for module M3 is Day, People, List(Day), List(People), List(List(Day)), ...

Module M3 forms a complete Gödel program on its own. Typical Gödel goals for this program are as follows.

```
<- Append3(Cons(Monday,Nil),Cons(Tuesday,Nil),
                                      Cons(Wednesday,Nil), x).

<- Append3(x,y,z,Cons(Fred,Cons(Bill,Cons(Mary,Nil)))).
```

The type system presented in this section can be regarded as a minimal one – it isn't hard to imagine a more complex type system, but it would be very hard to have less, since many-sortedness and polymorphism seem essential. Our experience with Gödel strongly suggests that this type system is, in fact, useful in practice and rarely is the need for a more complex type system found to be a pressing one, although there certainly are applications where having order-sortedness, for example, would be an advantage. Furthermore, this type system has the advantage that, under a rather general condition (see (HL91)), no run-time type checking is needed.

5. The Gödel Module System

Lists are a common data structure. Thus Gödel provides, via the module system, a built-in constructor for list processing. The constructor is called `List` and has arity 1. There is also a constant `Nil` and a function `Cons`, whose declarations are as follows.

```
CONSTANT    Nil : List(alpha).
FUNCTION    Cons : alpha * List(alpha) -> List(alpha).
```

The way the constructor `List/1`, the list notation, and the various list processing predicates are made available for use is by means of the module `Lists`, which is provided by the system. In the module `Lists`, the constructor `List/1` is declared, as are the constant `Nil` and the function `Cons`. Also a collection of useful list processing predicates, including `Append`, is defined there. Another module makes these predicates available for use by means of an `IMPORT` module declaration. For example, in module `M4`, the `IMPORT` declaration imports into `M4` the constructor `List`, the constant `Nil`, the function `Cons`, and various predicates including `Append`. Thus `List`, `Nil`, `Cons`, and `Append` are available for use in `M4`. In general, if a module imports from another module, it imports *all* the symbols exported by the other module.

With this use of modules, we can now give the final version of module `M1`, which is module `M4`. Modules `M4`, `Lists`, and `Integers`, which is imported by `Lists`, together form a Gödel program. Typical Gödel goals for this program are then as follows (where advantage is taken of the usual syntactic sugar which Gödel provides for lists).

```
<- Append3([Monday],[Tuesday],[Wednesday],x).

<- Append3(x,y,z,[Fred,Bill,Mary]).
```

In general, modules consist of two parts, a local part and an export part. The *local part* of a module is indicated by a `LOCAL` or `MODULE` module declaration. The *export part* of a module is indicated by an `EXPORT` or `CLOSED` module declaration. In these declarations, the keywords `LOCAL`, `MODULE`, `EXPORT` and `CLOSED` are followed by the name of the module. In fact, a module may have a local and an export part, or just a local part, or just an export part.

The other kind of module declaration is the `IMPORT` declaration, which has the form

```
MODULE      M4.

IMPORT      Lists.

BASE        Day, People.

CONSTANT    Monday, Tuesday, Wednesday, Thursday, Friday,
            Saturday, Sunday : Day;
            Fred, Bill, Mary : People.
PREDICATE   Append3 : List(alpha) * List(alpha) * List(alpha) *
                      List(alpha).

Append3(x,y,z,u) <-
            Append(x,y,w) &
            Append(w,z,u).
```

```
IMPORT      Name.
```

where `Name` is the name of a module.

The export part of a module begins with an `EXPORT` or `CLOSED` declaration, and contains zero or more `IMPORT` declarations, language declarations, and control declarations. The local part of a module begins with a `LOCAL` or `MODULE` declaration, and contains zero or more `IMPORT` declarations, language declarations, control declarations, and statements. If a module consists *only* of a local part, then this is indicated by using a `MODULE` declaration instead of a `LOCAL` declaration. The use of a `CLOSED` declaration instead of an `EXPORT` declaration means that the module allows some propositions or predicates declared in the module, which although in principle have a definition via Gödel code, to be actually implemented by another method. All system modules are closed modules.

In a similar way to the type system, Gödel has what is very much a minimal module system. However, our experience with the language supports the contention that, even though it is simple, it nearly always provides sufficient flexibility to do what is needed.

6. Gödel Meta-Programming Facilities

In the design of Gödel, particular attention has been paid to its meta-logical facilities. As has already been explained, there are two representations used for meta-programming. These are the non-ground and the ground representation. The non-ground representation is useful for the vanilla interpreter and various enhancements of it. However, the non-ground representation does not give sufficient flexibility for

Object		Meta	
Parameter	`alpha`	Term	`Par(6)`
Base	`Day`	Constant	`Day'`
Constructor	`List`	Function	`List'`
Variable	`x`	Term	`Var(3)`
Constant	`Monday`	Constant	`Monday'`
Function	`Cons`	Function	`Cons'`
Predicate	`Append`	Function	`Append'`
Connective	`<-`	Function	`<-'`
Quantifier	`ALL`	Function	`ALL'`

Fig. 1. A simplified ground representation

the manipulation of the representations of object level expressions, especially those containing variables. In contrast, using the ground representation, we can write in a declarative way many important kinds of meta-programs, such as program transformers, compilers, debuggers, abstract interpreters, program synthesizers, and theorem provers. We now turn attention to the ground representation, which is directly supported by Gödel.

The ground representation is a scheme for representing object languages, programs, modules, goals, theories, declarations, terms, and so on, as terms in a meta-language. The main idea is to represent an object program by a ground term at the meta-level. Figure 1 illustrates part of a (simplified) ground representation. The key aspect is that an object level variable is represented by a ground meta-level term. Note also that an object-level predicate is represented by a function at the meta-level. In fact, in Gödel, the details of the ground representation are not made explicit. (For details of various ways of setting up a ground representation, see (BK82), (HL89), or (HL88).) Instead, following an abstract data type approach, Gödel provides, via the system modules `Language`, `Program`, and `Theory`, a set of meta-predicates which allow a meta-program to access and manipulate terms representing object expressions. The total number of predicates provided by these three modules is around 160. The constants and functions actually used in the representation are mostly hidden in the local parts of these modules. However, we emphasize strongly that, even though the details of the ground representation are hidden, all the predicates exported by these modules can be given declarative definitions.

Gödel supports the ground representation of both (object) Gödel programs and (object) theories. The first of these is given by the module `Program` and the second by the module Theory. For various example meta-programs using these modules, we refer the reader to (HL91). We now discuss the module Language which provides some facilities for the manipulation of the representations of (object) languages common to both of these ground representations.

First, Language imports all the symbols exported by the modules Integers, Lists, and Strings. It also declares (amongst others) the following bases, which are required by the ground representation of languages.

```
BASE        OLanguage, OName, OFormula, OTerm.
```

OLanguage is the type of a term representing a language, OName is the type of a term representing the name of a symbol, OFormula is the type of a term representing a formula, and OTerm is the type of a term representing a term.

Nearly all predicates in Language have a first argument of type OLanguage. Since a term of type OLanguage represents an (object) language, such a term contains the representation of all the declarations in a language. All (object) languages admit the connectives &, \/, ~, <-, ->, and <->, and the quantifiers SOME and ALL. In addition, the language used by an (object) Gödel program admits IF-THEN-ELSE constructs and commits.

We now discuss in some detail a representative selection of predicates provided by Language.

The first two predicates are concerned with the representation of the connectives and quantifiers.

```
PREDICATE  And : OLanguage * OFormula * OFormula * OFormula;
           All : OLanguage * List(OTerm) * OFormula * OFormula.
```

And is intended to be true when its first argument is the representation of a language, its second and third arguments are the representations of formulas in this language, and its fourth argument is the representation of the formula which is the conjunction of the formulas in the second and third arguments. All is intended to be true when its first argument is the representation of a language, its second argument is the list of the representations of variables, its third argument is the representation of a formula in this language, and its fourth argument is the representation of the formula which is obtained by taking the universal quantification over the set of variables in the second argument of the formula in the third argument.

The next predicate is concerned with expressions independent of the object language.

```
PREDICATE  Variable : OTerm.
```

Variable is intended to be true when its argument is the representation of a variable. Variable covers the meta-logical uses of Prolog's *var*.

The next two predicates are concerned with ensuring terms and formulas can be expressed in a language and have certain properties.

```
PREDICATE  NonVarTerm : OLanguage * OTerm;
           GroundAtom : OLanguage * OFormula.
```

`NonVarTerm` is intended to be true when its first argument is the representation of a language and its second argument is the representation of a non-variable term in this language. `NonVarTerm` covers the meta-logical uses of Prolog's *nonvar*. `GroundAtom` is intended to be true when its first argument is the representation of a language and its second argument is the representation of a ground atom in this language.

The final predicate is useful for pulling apart and constructing terms.

```
PREDICATE  FunctionTerm : OLanguage * OTerm * OName * List(OTerm).
```

`FunctionTerm` is intended to be true when its first argument is the representation of a language, its second argument is the representation of a term in this language with a function at the top-level, its third argument is the representation of the name of this function, and its fourth argument is the list of representations of the top-level subterms of this term.

The module `Program` declares (amongst others) the following base.

```
BASE       OProgram.
```

`OProgram` is the type of a term representing an (object) Gödel program.

We now discuss in some detail a representative selection of predicates provided by `Program`. The first of the predicates provided by `Program` which we discuss here is `ProgramLanguage`, which has the following declaration.

```
PREDICATE  ProgramLanguage : OProgram  * OLanguage.
```

`ProgramLanguage` is intended to be true when its first argument is the representation of a program and its second argument is the representation of the language of this program.

The next predicate is concerned with accessing statements in open modules.

```
PREDICATE  StatementInModule : OProgram * String * OFormula.
```

`StatementInModule` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, and its third argument is the representation of a statement in this module.

The next predicate is needed for dynamic meta-programming.

```
PREDICATE  InsDelStatement : OProgram * String * OFormula * OProgram.
```

`InsDelStatement` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, its third argument is the representation of a statement in the language of this module wrt this program, and its fourth argument is the representation of a program which differs from the program in the first argument only in that it also contains this statement in this part of this module. **InsDelStatement** can be used to insert a statement by calling it with the first three arguments instantiated and the fourth a variable. It can be used to delete a statement by calling it with the first argument a variable and the remaining arguments instantiated.

The final predicate is concerned with running goals for a program.

```
PREDICATE  Succeed : OProgram * OFormula * OTermSubst.
```

Succeed is intended to be true when the first argument is the representation of a program, its second argument is the representation of a goal in the language of this program, and its third argument is the representation of a computed answer from an SLDNF-refutation (using the Gödel computation rule) for this goal and this program. The predicate **Succeed** is the Gödel equivalent of the standard *demo* predicate.

The ground representation can also be used for representing object (many-sorted) theories, theorems, declarations, terms, and so on, as terms in a meta-language. A theory is like the local part of a module having only a local part, except that it does not contain IF-THEN-ELSE constructs, commits, or control declarations, and it contains arbitrary first order formulas instead of statements. A theory can import modules. A typical application for the system module **Theory** could be a program synthesis system. For such a system, the specification for a program would be expressed as a theory, which perhaps imported some modules. Then a meta-program would perform transformation on this theory until it took the form of (essentially) a Gödel module. After further modification (for example, adding control information and/or employing conditionals), the synthesized program would result.

7. Other Gödel Facilities

We discuss Gödel's control facilities. Prolog uses the computation rule which always selects the leftmost literal. In contrast, Gödel has a flexible computation rule, which may select a literal other than the leftmost literal. The advantages of a flexible computation rule are well known. These are that it can be used to ensure safeness (especially of negative calls), assist termination, assist efficiency, and control pruning. The Gödel computation rule is partly specified by means of DELAY control declarations, which are syntactic variants of the **when** declarations of NU-Prolog (TZ88) and cause certain calls to be delayed until they are sufficiently instantiated. Many Gödel system predicates have DELAY declarations. Furthermore, a programmer can specify co-routining behaviour by means of these declarations.

Gödel has constraint solving capabilities in the domains of integers and rationals. Gödel can solve systems of (not necessarily linear) constraints which involve integers, variables which range over bounded intervals of integers, and the usual functions and predicates with integer arguments. It can also solve systems of linear rational constraints involving rationals, variables ranging over the rationals, and the usual functions and predicates with rational arguments.

The Gödel pruning operator is based on the commit of the concurrent languages. In fact, the simplest form of the Gödel commit is similar to the commit of the concurrent languages. Another form is the one solution operator. Each of these is a special case of the most general form of the commit (HL91), which has the property that programs containing the commit are closed under the usual program transformations, such as unfolding.

In addition to the motivation for Gödel discussed earlier, we make two further points. The first is that because Gödel has greatly improved declarative semantics compared to Prolog it is eminently suited as a teaching language. The other point is that, even if Gödel itself never becomes widely used, we hope some of its ideas will be adopted by other languages. Significant extensions and variations of Prolog have been introduced and studied by the logic programming community over the last 15 years. Unfortunately, these languages have been essentially built on top of Prolog and therefore inherit many of Prolog's semantic problems. For example, most of these languages use Prolog's approach to meta-programming. We hope that the designers of current and future logic programming languages will see from Gödel that it really *is* possible to design and implement a language with a declarative semantics that we don't have to be embarrassed about.

8. Discussion

Much impressive research has been carried out in the design of logic programming languages over the last 20 years. As well as the original Prolog language, we now have numerous extensions and variations, such as concurrent, meta-logical, constraint, higher-order, and object-oriented programming. On the other hand, probably 90% of all logic programs have been written for Prolog systems more or less functionally equivalent to the first Marseille Prolog system which we now know to have serious design deficiencies! This is a rather disappointing situation. Modern Prolog systems are many orders of magnitude more efficient than the original Marseille version, but from a language design point of view are essentially the same. It is hard to escape the conclusion that researchers in this area have spent too much time worrying about efficiency and parallelism and too little time worrying about whether their languages have even the most basic software engineering facilities and have any real credibility at all as declarative languages! It is long past the time for logic programmers to concentrate more on basic design issues.

First, we must accept that a type system and a module system are absolutely essential in a modern language. The arguments for these are overwhelming. Second, we must avoid the Prolog constructs (mainly the meta-logical ones) which so severely compromise the declarative semantics of programs. The techniques for this are now well understood, although further research on their efficient implementation is needed since this problem has been greatly neglected. In particular, new languages which take (full) Prolog as their basis cannot be accepted any more. Starting from a solid foundation of a core language with a type and a module system and declarative meta-logical facilities, we need to explore the various interesting extensions, such as constraint, higher-order, object-oriented, concurrent, and so on.

So what are the most interesting directions to pursue for the immediate future of logic programming language design? For concreteness, we discuss various possible extensions of Gödel, which can be considered to be an acceptable core language.

The current definition of Gödel includes some modest constraint-solving capabilities. However, it would be interesting to extend these capabilities to include the more sophisticated constraint-solving capabilities of established constraint languages. The implementation techniques for this are now well established, although more work is needed on designing clean and elegant facilities at the language level.

It is easy to conceive of a language, which would bear the same relation to Gödel as the concurrent logic programming languages have to Prolog. The main changes would be to drop Gödel's language features which depend on negation, enforce the use of the commit in every statement, and extend the control facilities to give the control available in the concurrent languages. Furthermore, recently, there have been attempts to build languages which include both the Prolog style of computation and the concurrent style, as, for example, with the Andorra Prolog language and Andorra computational model (HB88), (SWY91a), (SWY91b), (Nai88). If such a unification is possible for Prolog-based languages, then it is equally possible for Gödel.

A major trend in programming languages, not just logic programming languages, in recent years has been the introduction of object-oriented features. The kind of type system already discussed provides some of the features required. However, some significant extensions are also needed. These include an order-sorted type system (for handling subtypes), parametric modules, and specialised facilities for structuring theories which represent objects.

Making a higher-order extension of Gödel would involve applying to Gödel the implementation techniques for adding higher-order features which have been developed for Prolog. The addition of higher-order facilities to Gödel would fill a gap in the non-ground representation, for which only quantifier-free formulas can be represented. With higher-order facilities, quantified formulas can be represented using λ-terms.

The extensions discussed so far (constraint, concurrent, object, and higher-order) are to a large extent orthogonal to one another. Thus, it ought to be possible to build an extension of Gödel which includes all four of these in a simple and elegant way.

Another extension is to add other meta-programming modules. The module `Program` provides a representation for object Gödel programs and is the most important ground representation. Gödel also currently provides the module **Theory** for theorem proving and program synthesis applications. However, we also need meta-programming modules for representing programs from other programming languages. For example, it may be desirable to write in Gödel a program transformer for Prolog programs. For this, we need a version of **Program** for representing Prolog programs. In fact, all these variations of **Program** will be easy to implement once the difficulties of implementing **Program** itself have been overcome. In the longer term, a more flexible approach to providing ground representations may be preferable. Instead of building in a fixed set of ground representations, lower-level primitives would be provided which programmers could use to define their own ground representations.

Most logic programming systems make use of some kind of "closed world assumption". For example, Gödel uses the completion of the program as the theory. The newer semantics currently being studied, such as the well-founded semantics, are also based on such an assumption. While this assumption is justified in many circumstances, there are also many for which it is not justified, especially for sophisticated knowledge base applications. Consequently, it is desirable to allow logic programming systems to use more general theories than completions and to have corresponding proof procedures for theorem proving in such theories. In any case, the argument for a "closed world assumption" has always been somewhat suspect since it is based largely on a desire to justify the use of the efficient proof procedure SLDNF-resolution! However, with our better understanding of theorem-proving implementation techniques and with much more powerful computers becoming available, the efficiency argument for SLDNF-resolution (and related proof procedures) looks increasingly untenable. If we want logic programming systems to be more widely applicable, we really must look seriously at this issue.

Finally, let us consider longer term matters. Program synthesis, the synthesis of an efficient program from a specification, is one of main goals of Computer Science. It should also be a major goal of logic programming. In fact, logic programming has already made substantial progress in this direction. The very idea that a theory can be a program at all is a contribution of logic programming. Furthermore, some logic programming systems have raised the level of a program toward a specification. For example, NU-prolog has a control preprocessor which automatically adds useful control declarations to a large class of programs. It also admits a larger class of programs than most systems, allowing arbitrary formulas in the bodies of goals and clauses. (Gödel also has these facilities.) However, much more is possible, as is clear from the large amount of research taking place on program synthesis outside logic programming.

A general framework for program synthesis is as follows. First, the specification is written down. This is either a first order theory, as specifications are usually given, or an intended interpretation. (The choice here is between giving the intended interpretation directly or else giving a theory which has the intended interpretation as a model.) Assuming the latter case, the specification is then transformed (soundly) into a more efficient theory. This is the hard part and techniques already developed by researchers in program synthesis will be needed here, plus many others probably. Notice, however, that program synthesis is being formulated here as a problem of program *transformation*. The specification and the program are (essentially) in the same language – the main difference is that the program can be run much more efficiently under some suitable proof procedure. The final, and easier, step is to add control declarations, pruning, if-then-else constructs, and so on, to obtain the final efficient program. Program synthesis is a challenging problem. However, when we have solved it, we truly will have achieved declarative programming!

9. Conclusion

Logic programming has a credibility problem, which arises from the large gap between theory and practice. Put simply, the problem is that up to now there have been no practical logic programming languages which also have a satisfactory semantics, especially declarative semantics. Thus logic programming has not delivered what it so clearly promises – practical, declarative programming. As a result, the field has not had the impact on computing that one could have expected by now (although there have been some successes, notably in databases, expert systems, and natural language processing). What is needed to build practical, declarative logic programming languages has been known for a number of years – there only remains the effort and commitment on the part of enough researchers to bring these good ideas to life.

Acknowledgement

Much of the material of this paper is adapted from the Gödel report, which was written in collaboration with P.M. Hill.

References

[BK82] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 153–172, Academic Press, 1982.

[Col90] A. Colmerauer. An introduction to Prolog III. In J.W. Lloyd, editor, *Proceedings of the Symposium on Computational Logic*, Brussels, pages 37–79, Springer-Verlag, 1990.

[DvHS*88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, pages 693–702, 1988.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[HB88] S. Haridi and P. Brand. Andorra Prolog: an integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, pages 745–754, 1988.

[HL88] P.M. Hill and J.W. Lloyd. *Meta-Programming for Dynamic Knowledge Bases*. Technical Report CS-88-18, Department of Computer Science, University of Bristol, 1988.

[HL89] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In H.D. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–52, MIT Press, 1989. Proceedings of the Meta88 Workshop, June 1988.

[HL91] P.M. Hill and J.W. Lloyd. *The Gödel Report*. Technical Report TR-91-02, Department of Computer Science, University of Bristol, 1991. Revised September 1991.

[HLS90] P.M. Hill, J.W. Lloyd, and J.C. Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, Munich, 1987.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[Nai88] L. Naish. Parallelizing NU-Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, pages 1546–1564, MIT Press, 1988.

[NM88] G. Nadathur and D. Miller. An overview of λ-Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, pages 810–827, MIT Press, 1988.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[SWY91a] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I engine : a parallel implementation of the Basic Andorra model. In K. Furukawa, editor, *8th International Conference on Logic Programming*, Paris, pages 825–839, MIT Press, 1991.

[SWY91b] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor: supporting full Prolog on the Basic Andorra model. In K. Furukawa, editor, *8th International Conference on Logic Programming*, Paris, pages 443–456, MIT Press, 1991.

[TZ88] J. A. Thom and J. Zobel. *NU-Prolog Reference Manual, Version 1.3*. Technical Report, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

EFFICIENT BOTTOM-UP EVALUATION OF LOGIC PROGRAMS

RAGHU RAMAKRISHNAN, DIVESH SRIVASTAVA and S. SUDARSHAN
Computer Sciences Department
University of Wisconsin – Madison
Madison WI 53706, U.S.A.

Abstract. In recent years, much work has been directed towards evaluating logic programs and queries on deductive databases by using an iterative bottom-up fixpoint computation. The resulting techniques offer an attractive alternative to Prolog-style top-down evaluation in several situations. They are sound and complete for positive Horn clause programs, are well-suited to applications with large volumes of data (facts), and can support a variety of extensions to the standard logic programming paradigm.

We present the basics of database query evaluation and logic programming evaluation, and then discuss bottom-up fixpoint evaluation. We discuss an approach based upon using a program transformation ("Magic Templates") to restrict search, followed by fixpoint computation using a technique ("Semi-naive evaluation") that avoids repeated inferences. The program transformation technique focuses the fixpoint evaluation, which is a forward-chaining strategy, by propagating bindings in the goal in a manner that essentially mimics the backward-chaining behavior of top-down evaluation strategies.

After presenting the basic framework for bottom-up evaluation, we discuss several refinements that optimize the treatment of non-ground terms, improve memory utilization, reduce the cost of duplicate checking, and utilize the declarative semantics of the program to perform semantic query optimization in a number of ways. We also consider several extensions to the logic programming paradigm, and discuss how the bottom-up evaluation framework can be used to support these extensions. The extensions include support for negation, set-terms, constraint manipulation and quantitative reasoning.

Finally, we discuss several systems based upon bottom-up evaluation, including Aditi (Univ. of Melbourne), EKS-V1 (ECRC-Munich), Glue/NAIL! (Stanford Univ.) and LDL (MCC-Austin). We have developed such a system, called CORAL, and we present this in more detail.

Key words: Logic Programs, Bottom-Up Evaluation, Deductive Databases, Database Query Languages.

1. Introduction

Traditionally, logic programming language implementations have evaluated programs top-down. Recent developments in evaluation of database queries have provided a means of evaluating logic programs bottom-up while still retaining the focusing properties of top-down evaluation, based in large part on optimizing program transformations.

EXAMPLE 1.1. As a simple example of the power of the bottom-up approach, consider a definition of transitive closure

$$\begin{array}{lll} t(X,Y) & :- & e(X,W), t(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(5,Y). \end{array}$$

If e is cyclic, Prolog will not terminate on this program. Even if e is not cyclic, there are values for e such that Prolog takes time exponential in the size of the answer.

To evaluate the original query bottom-up, we first obtain the following program by first applying the Magic Templates[1] transformation (Section 3.1) and then predicate factoring (Section 5.1):

$$\begin{array}{lll}
magic\_t^{bf}(W) & :- & ft(W). \\
magic\_t^{bf}(5). & & \\
ft(Y) & :- & magic\_t^{bf}(X), e(X,Y). \\
query(Y) & :- & ft(Y).
\end{array}$$

Evaluating this program using Semi-naive bottom-up evaluation (Section 3.2) always terminates and computes the answer to the query in time linear in the answer size.

Essentially the same program results if Magic Templates plus factoring is applied to versions of the transitive closure expressed with different forms of the recursive rule, including the left-recursive and binary recursive forms (upon which Prolog does not terminate for any values of e). This example is presented in detail in Section 5. □

This is an example where the bottom-up approach based on program transformations has notable success. Even on programs for which the transformations do not achieve such large gains, the Magic Templates transformation ensures that no "irrelevant" goals or facts are generated, and in many cases significant advantages accrue when bottom-up evaluation is used, primarily because:

- With memoing, it implements a form of dynamic programming, which eliminates a great deal of redundant computation, and
- It is sound and complete, and thus the declarative semantics of the program (which is taken to be its least Herbrand model) is preserved. This non-procedural approach allows for much more flexibility in program transformation and optimization.

In this paper, we give a tutorial introduction to the bottom-up approach to logic program evaluation. We introduce notation and preliminary definitions in Section 2, and present a summary of the bottom-up evaluation method based on the Magic Templates program transformation and Semi-naive fixpoint evaluation in Section 3. In this section, we also discuss some alternatives to the Magic Templates approach and compare top-down and bottom-up methods briefly. We present a survey of some related program transformations used in the bottom-up approach in Section 5. In Section 6, we discuss various extensions to the Horn clause logic programming

[1] Earlier versions of the algorithm were called Magic Sets. See Section 3 for details.

paradigm. We describe some systems based upon bottom-up evaluation in Section 7, and discuss one of them (CORAL) in more detail in Section 8.

We note that some related evaluation techniques (e.g., Vieille's QSQR and QoSaQ [108; 109; 110], parallel evaluation techniques (e.g., work by Gonzalez-Rubio, Bradier and Rohmer [35], Ganguly, Silberschatz and Tsur [32], Cohen and Wolfson [24], and others, e.g. [29; 74; 104; 112]), work on integrity constraints (e.g., Bry, Decker and Manthey [15], Vieille et al. [106] and others), work on production rule systems (e.g., Forgy [30], Sellis, Lin and Raschid [92], Widom et al. [111], and others, and systems such as OPS 5), work on intelligent query answering (e.g. [54; 22]) etc. are not discussed in this tutorial presentation due to lack of space. Our coverage attempts to present one coherent set of results that are indicative of the field, rather than a survey, and is doubtless influenced by our personal perspectives.

2. Notation and Preliminary Definitions

The language considered in this paper is that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols. A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A tuple of terms is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$.

A substitution is an idempotent mapping from the set of variables of the language under consideration to the set of terms. Substitution operations are usually written in postfix form. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \sigma[\varphi]$. Substitutions are denoted by lower case Greek letters θ, σ, ϕ, etc. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $t_1[\sigma] = t_2[\sigma]$. σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier that is unique up to renaming of variables.

A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body*. A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $\langle P, Q\rangle$ where P is a set of predicate definitions and Q is the *input*, which consists of a query, or goal, and possibly a set of facts for "database predicates" appearing in the program.

We follow the convention in deductive database literature of separating the set of rules with non-empty bodies (the set P) from the set of facts, or unit clauses, which appear in Q and are called the *database*. P is referred to as the *program*, or the set of rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. The meaning of a logic program is given by its least Herbrand model [102].

Following the syntax of Edinburgh Prolog, definite clauses (rules) are written as

$$p :- q_1, \ldots, q_n.$$

read declaratively as q_1 and q_2 and ...and q_n implies p. Names of variables begin with upper case letters, while names of non-variable (i.e., function and predicate) symbols begin with lower case letters. In addition, the following notation is used for lists: the empty list is written [ ], and a list with head H and tail L is written $[H|L]$. Note that control primitives such as Prolog's cut are not allowed.

3. The Bottom-Up Approach

The bottom-up approach that we consider consists of a two-part process. First, the program is rewritten in a form so that the bottom-up fixpoint evaluation of the program will be more efficient; next, the fixpoint of the rewritten program is computed by bottom-up iteration. Subsection 3.1 describes the initial rewriting, while Subsection 3.2 investigates the computation of the fixpoint of the rewritten program. Both these steps can be refined further; this is discussed in later sections.

3.1. The Magic Templates Rewriting Algorithm

As described in [13; 73], the initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or sips. For each rule, the associated sip determines the order in which the body literals are evaluated. In analogy to top-down evaluation, we can think of rules being used to generate answers to queries on the head predicate, with some arguments bound to given terms; different sips can be chosen for each head binding pattern.

We present a simplified version of the Magic Templates algorithm, tailored to the case that sips correspond to left-to-right evaluation with all arguments considered "bound" (perhaps to a free variable), as in Prolog. The idea is to compute a set of auxiliary predicates that contain the goals.

The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples.

DEFINITION 3.1. **The Magic Templates Algorithm**
Let P be a program, and $? - q(\bar{c})$ a query on the program. We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. Create a new predicate *magic_p* for each predicate p in P. The arity is that of p.
2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body.
3. For each rule r in P with head, say, $p(\bar{t})$, and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body contains the literal $magic\_p(\bar{t})$, and all literals that precede q_i in the rule.
4. Create a *seed* fact $magic\_q(\bar{c})$ from the query.

□

EXAMPLE 3.1. Consider the following program.

$$\begin{array}{ll}
sg(X,Y) & :-\ flat(X,Y). \\
sg(X,Y) & :-\ up(X,U), sg(U,V), down(V,Y). \\
?-sg(john,Z) &
\end{array}$$

For a choice of sips that orders body literals from left to right, as in Prolog, the Magic Templates algorithm rewrites it as follows:

$$\begin{array}{ll}
sg(X,Y) & :-\ magic\_sg(X,Y), flat(X,Y). \\
sg(X,Y) & :-\ magic\_sg(X,Y), up(X,U), sg(U,V), \\
 & \qquad down(V,Y). \\
magic\_sg(U,V) & :-\ magic\_sg(X,Y), up(X,U). \\
magic\_sg(john,Z). &
\end{array}$$

□

We present some results that characterize the transformed program P^{mg} with respect to the original program P, from [73]. The following theorem ensures soundness.

THEOREM 3.1. *[73] $\langle P,Q\rangle$ is equivalent to $\langle P^{mg},Q\rangle$ with respect to the set of answers to the query.*

DEFINITION 3.2. Let us define the *Magic Templates Evaluation Method* as follows:

1. Rewrite the program $\langle P,Q\rangle$ according to the choice of sips using the Magic Templates algorithm.

2. Evaluate the fixpoint of the rewritten program.

□

We hope that the slight abuse of notation in having the same name for the evaluation method and the rewriting algorithm will not lead to confusion; the distinction should be clear from the context. The second step above is presented in more detail in the next subsection. The rewriting has the important effect of mimicking Prolog in that (modulo optimizations such as tail recursion optimization and intelligent backtracking) only goals and facts generated by Prolog are generated. [2]

The careful reader will notice that some joins are repeated in the bodies of rules defining magic predicates and modified rules. The *supplementary* version of the rewriting algorithm essentially identifies these common subexpressions and stores them (with some optimizations that allow us to delete some columns from these intermediate, or supplementary, relations). We refer the reader to [13] for details.

Magic Templates is often presented along with an adornment rewriting that annotates predicates with a string composed of characters 'f' and 'b', with one character for each argument. This step, along with a modification of Magic Templates rewriting that projects out of magic predicates those arguments that have an f adornment, is used to ensure that the rewritten program generates only ground facts. For brevity, we omit this step.

3.2. Iterative Fixpoint Evaluation

We describe a refinement of bottom-up fixpoint evaluation called Semi-naive fixpoint evaluation, which is cited as the method of choice in the deductive database literature. Derivations made in one iteration are not repeated in subsequent iterations since in each iteration, only rule instantiations that utilize at least one new fact (i.e., one generated for the first time in the previous iteration) are considered.

We follow the presentation in [51] in the rest of this section, with some simplifications.

Let us first define a binary operator W_P, whose role is similar to that of the well-known T_P operator of [102]. We use square brackets ([...]) to denote multisets:

$$\begin{aligned} W_P(X,Y) = [h[\theta] \mid \; & h :- b_1,\ldots,b_k \text{ is a rule of } P, \\ & \theta \text{ is mgu of } (b_1,\ldots,b_k) \text{ and } (d_1,\ldots,d_k), \\ & Y \subseteq X, \{d_1,\ldots,d_k\} \subseteq X, \text{ and} \\ & k \geq 0 \text{ and } \{d_1,\ldots,d_k\} \cap Y \neq \emptyset.] \end{aligned}$$

[2] This is strictly true only for programs that do not generate non-ground facts. The method can be modified slightly to make the claim valid for all programs and to allow tail recursion optimization. For details, see [81].

The multiplicity of an element $h[\theta]$ is determined by the number of distinct rule instantiations of which it is the head literal. A more rigorous definition that formally accounts for the multiplicity is given in [51]; we have chosen to use an informal presentation for ease of exposition. Intuitively, W_P only allows deductions from the set of facts X that use the "new" facts Y, and "counts" a fact as often as it is derived (using distinct rule instantiations). We now define Semi-naive iteration. In the following definition, *set* is an operator that takes a multiset and returns a set, and *subs* is an operator that takes a multiset and returns an irredundant set. (An irredundant set, or *irrset*, is a set of elements such that no element subsumes another.)

DEFINITION 3.3. **Semi-naive Iteration (SN)**
Let $S_{-1} = S_0 = \delta_0 = \mathcal{F}$.

$$\begin{aligned}
\delta_{n+1} &= \mathit{dup\_elim}(W_P(S_n, \delta_n - S_{n-1})) \\
S_{n+1} &= \mathit{dup\_elim}(S_n \cup \delta_{n+1}) \\
S &= \lim_{n\to\infty} S_n \\
GC &= \mathit{set}(S)
\end{aligned}$$

where $\mathcal{F}$ is the set of facts, or rules with empty bodies, in the program P, and *dup_elim* is either *set* or *subs*. We refer to the variant with $\mathit{dup\_elim} = \mathit{set}$ as SN_S, and the variant with $\mathit{dup\_elim} = \mathit{subs}$ as SN_I. □

In Semi-naive iteration, the set of facts produced in iteration n (δ_n) is compared with the set of known facts (S_{n-1}) to identify the new facts produced ($\delta_n - S_{n-1}$). Duplicates generated within the same iteration are eliminated by the *set* operation. Only derivations that use one of these new facts are carried out in iteration $n + 1$. This avoids generating many duplicate facts by avoiding repeated derivations. The algorithm terminates (at step $n+1$) when $S_{n+1} = S_n$; we must test whether $\delta_{n+1} \subseteq S_n$. Consequently, Semi-naive iteration terminates if and only if S is finite.

In this presentation we ignore the issue of how to ensure that only derivations that use a new fact are carried out. This is addressed in several papers (e.g. [6; 9; 5]). The problem becomes more complicated when it is desired to apply rules in a specified order in each iteration. This issue is discussed in [49; 78; 90].

The set GC is the set of *generated consequences* of the program.

Let *ground* be an operator that takes a set of possibly non-ground facts and a domain $\mathcal{D}$ and returns the set of ground facts containing only constants from $\mathcal{D}$ that are instances of the input set. The following result shows that the above iterative methods are consistent with the usual least Herbrand model semantics of [102]; here implicitly $\mathcal{D}$ is the domain for the program in question. We denote the least Herbrand model of the program by $\mathcal{M}$.

PROPOSITION 3.1. *[51] The set of generated consequences GC of a program computed using Semi-naive Iteration (either SN_S or SN_I) is such that $ground(GC) = \mathcal{M}$.* □

We note that ordinary fixpoint evaluation — frequently called "Naive" (N) iteration in the deductive database literature — corresponds to the case where the second argument of W_P in the definition of Semi-naive Iteration is replaced by S_n.

EXAMPLE 3.2. The following program illustrates the difference between N and SN.

```
d :- c.
c :- b.
b :- a.
a.
b.
```

Note that SN_S and SN_I behave identically on this program. Both N and SN generate a and b in the first iteration. In the second iteration, SN generates c and a second occurrence of b, and eliminates the duplicate b. In the next iteration, it generates only d, since c was the only new fact found in the second iteration. In contrast, N additionally generates in each iteration all the facts that it generated in previous iterations. □

3.3. Related Work

The Alexander method was proposed independently of the Magic Sets approach in [83]. It is essentially the *supplementary* variant of the Magic Templates method, described in [13]. Seki has generalized the method to deal with non-ground facts and function symbols, and has called the generalized version Alexander Templates [91]. The Alexander methods always use a single left-to-right sip for each rule, for all possible goals.

The Magic Templates idea was developed in a series of papers ([7; 13; 73]). Several variants of the Magic Templates idea have also been proposed. For example, it is possible to compute supersets of the magic sets without compromising soundness. Although this results in some irrelevant computation, it may be possible to compute supersets more efficiently than the magic sets themselves [94].

Although the Magic Templates idea was introduced to deal with recursion, it provides significant improvements for non-recursive queries as well. In [56; 57] it is shown that the technique can be extended to deal with SQL programs, including those containing features like group-by, aggregation and arithmetic conditions. In [55] a performance evaluation, carried out on a DB2 relational system, is presented, demonstrating that the technique performs comparably to standard database techniques, and is often significantly better.

The Magic and Alexander methods are based on program transformations. Other methods use a combination of top-down and bottom-up control to propagate bindings. Pereira and Warren presented a memoing top-down evaluation procedure based on Earley deduction [68]. This evaluation procedure may be viewed as a top-down evaluation procedure that incorporates memoing. Vieille has proposed a method called QSQ [108; 109; 110] that can be viewed as follows. Goals are generated with a top-down invocation of rules, as in Prolog. However, there are two important differences: 1) whenever possible, goals and facts are propagated set-at-a-time, and 2) all generated goals and facts are memoed. If a newly generated goal is already memoed, this is recognized by duplicate elimination. Dietrich and Warren have proposed a method called Extension Tables [28]. This method is very similar to QSQ, but performs computation tuple at a time.

Finally, Kifer and Lozinskii have proposed a method called Filtering, which is based on constructing a rule-goal graph [46; 47]. There is a node in the graph for each predicate, and for each rule, and arcs from predicate nodes to each rule node in whose body it appears, and from rule nodes to the predicates that they define. The idea is to compute the fixpoint by propagating tuples along these arcs, and to restrict the computation by attaching "filters" to arcs.

The reader is referred to [63] for a more detailed discussion of related work.

4. Top-Down Versus Bottom-Up

In the past, bottom-up methods have not been seriously considered for the evaluation of logic programs because of a serious drawback: no techniques were known that avoided computing an unbounded number of irrelevant facts. Now that such techniques are known, it is worth reconsidering bottom-up approaches. They offer three significant advantages: (1) The declarative least Herbrand model semantics is guaranteed for positive Horn clause programs. (2) Redundant derivations are avoided through memoing, leading to asymptotic gains in efficiency for programs in which goals or facts can be derived in many ways. (3) As a consequence of (1), no operational guarantees need be made, thereby making possible a number of semantic optimizations. One example of a powerful optimization made possible by the declarative semantics is *factoring*, discussed in Section 5.

There are a number of top-down evaluation techniques for logic programs such as Prolog, the Query-Subquery (QSQ) approach and its extensions [108; 110], and Extension Tables [28]. SLD-AL resolution is a theoretical model of the execution of top-down evaluation techniques that perform memoization of facts. Memoing top-down evaluations are similar to the bottom-up approach with respect to advantages (1), (2), and to some extent (3) above. A natural question is how bottom-up evaluation techniques compare with memoing as well as non-memoing top-down evaluation techniques.

Initial comparisons were based on the number of facts derived by the different techniques. Thus, Ramakrishnan [73; 74] presents a class of evaluations and shows that within this class bottom-up evaluation of a program rewritten using Magic Templates computes an optimal number of facts. Seki [91] presents a direct comparison between the set of facts computed using Alexander Templates rewriting, and using SLD-AL resolution. Bry [16] shows that several top-down and bottom-up evaluation techniques can be viewed as specializations of a technique called Backward Fixpoint Procedure. Thus the least fixpoints of all these techniques have the same set of facts. Given a bottom-up evaluation that does not repeat derivation steps, these comparisons can be extended to include the number of derivations made.

There are three limitations to the above results. The first is that the comparisons do not include the cost of each inference. The second is that these comparisons ignore optimizations that are routinely performed on Prolog programs such as tail-recursion optimizations (Section 5.2). And third, the comparisons assume that all answers are required, and do not provide enough insight for the case that only one answer is needed (although there is no change in the worst case comparison).

Ullman [99] addressed the first problem, and compared the time cost of evaluation of Datalog programs using bottom-up evaluation, and a particular top-down evaluation technique. This comparison showed that for the class of safe Datalog programs (i.e., only ground facts are computed and no uninterpreted function symbols are allowed), bottom-up evaluation can be performed with a cost that is no more than that of the top-down evaluation in order of magnitude. Ramakrishnan and Sudarshan [80] compared a model of Prolog evaluation that performs tail recursion optimization with a model of bottom-up evaluation that uses a rewriting due to Ross [85] that performs tail recursion optimization. They showed that with this model of bottom-up evaluation the number of inferences and the time cost are (in order of magnitude) no more than that of Prolog, for a class of programs that includes safe Datalog and that allows ground terms to be constructed using uninterpreted function symbols, but with some restrictions. The rewritten program may generate non-ground facts, although the original program is not allowed to do so (for the class of programs considered). For the general case, where arbitrary non-ground terms are allowed, there are several problems that increase the cost per inference. These problems are faced not just by bottom-up evaluation schemes, but also by top-down evaluation schemes that perform memoization of facts. These issues are discussed in Section 5.10.

5. Some Important Program Optimizations

In this section, we survey a number of optimizations that apply to positive Horn programs and guarantee query equivalence in the least fixpoint model. That is, the same set of answers according to the declarative semantics of logic programs is computed. Note that no further guarantees are offered: no equivalence is guaranteed with respect to other program predicates, nor is any ordering preserved in the generation of answers.

This survey is not intended to be comprehensive, but rather to provide an indication of the flexibility that is available when we only need to preserve the declarative semantics. Ensuring that the declarative semantics is preserved has long been viewed as a bar to efficient evaluation; the results surveyed here indicate that it can sometimes be considerably more efficient to guarantee this semantics, rather than an operational semantics. For lack of space we do not cover some approaches to query optimization such as the semantic query optimization techniques proposed by Minker, King and others (see e.g. [48; 17]).

5.1. Predicate Factoring

The basic idea behind predicate factoring is to replace a predicate by two predicates of strictly smaller arity. This can result in significant speedups, as the example in this section illustrates. We present a simplified description that is tailored to the case of programs obtained by applying the Magic Templates algorithm described in Section 3.1. Our presentation is through the use of examples, and we do not describe sufficient conditions for the optimization to apply in general. We refer the reader to [64] for a detailed treatment.

In essence, we seek to take advantage of the magic predicates to replace the original predicate in P^{mg} by its projection onto its f argument positions. Thus, the original (non-magic) predicate in P^{mg} is factored into the magic predicate, which computes the b argument positions, and this new predicate, which computes the f argument positions.

EXAMPLE 5.1. We begin with a familiar example, transitive closure. While efficient algorithms are known, the rewriting algorithms presented in [65] were the first to automatically derive unary programs for single-selection queries, for all three forms (left-linear, right-linear, non-linear) of the recursive rule. We achieve the same result here by first applying the Magic Templates transformation and then factoring the rewritten program. To illustrate the technique, we consider a single program that includes all three forms of the recursive rule, although any one would suffice. It should be evident that we would also obtain a unary program if the original program contained just one of the recursive rules. Consider the program and single-selection query below:

$$\begin{array}{lll}
t(X,Y) & :- & t(X,W), t(W,Y). \\
t(X,Y) & :- & e(X,W), t(W,Y). \\
t(X,Y) & :- & t(X,W), e(W,Y). \\
t(X,Y) & :- & e(X,Y). \\
query(Y) & :- & t(5,Y).
\end{array}$$

The Magic Templates algorithm rewrites this to:

$$\begin{array}{lll}
magic\_t^{bf}(W) & :- & magic\_t^{bf}(X), t^{bf}(X,W). \\
magic\_t^{bf}(W) & :- & magic\_t^{bf}(X), e(X,W). \\
magic\_t^{bf}(5). & & \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), t^{bf}(X,W), t^{bf}(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), e(X,W), t^{bf}(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), t^{bf}(X,W), e(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), e(X,Y). \\
query(Y) & :- & t^{bf}(5,Y).
\end{array}$$

If we identify $magic\_t^{bf}$ tuples with goals in a top-down evaluation, we see that only the last occurrence of t^{bf} in a rule body generates new goals, and further, the answer to a new goal is also an answer to the goal that invoked the rule. In fact, every answer to a subgoal is also an answer to the query goal $magic\_t^{bf}$. A second observation is that if c is generated as an answer to a subgoal, then a new subgoal $magic\_t^{bf}(c)$ is also generated. These observations lead us to conclude that it does not matter to which subgoal an answer corresponds; its role in the computation is the same in any case. That is, $t^{bf}(X,Y)$ can be factored into $bt(X)$ and $ft(Y)$ in the Magic program. Doing this and applying some simple syntactic optimizations, which are discussed in [64], we finally obtain the following unary program:

$$\begin{array}{lll}
magic\_t^{bf}(W) & :- & ft(W). \\
magic\_t^{bf}(5). & & \\
ft(Y) & :- & magic\_t^{bf}(X), e(X,Y). \\
query(Y) & :- & ft(Y).
\end{array}$$

□

5.2. Tail Recursion Optimization

An optimization that is closely related to factoring is tail-recursion optimization. Consider a program of the form

$$\begin{array}{l}
p(0,1). \\
p(X,Y) :- X > 0, p(X-1,Y).
\end{array}$$

If there is a query $? - p(10, X)$, queries $? - p(9, X)$ and so on till $? - p(0, X)$ are set up. At this point an answer $p(0,1)$ is generated by bottom-up evaluation of a program rewritten using Magic Templates. However a Prolog system can recognize that the second rule is "tail-recursive," and can avoid generating answers for intermediate goals. The reason is that any answer substitution for the last goal results in an answer for the head of this rule with the same answer substitution. Applying this optimization for each goal on this rule, when the goal $? - p(0, X)$ succeeds, an answer $p(10,1)$ is created directly, rather than an answer $p(0,1)$.

Ross [85] shows how the same optimization can be performed in bottom-up evaluation. The details are beyond the scope of this paper, but the essential idea is to generate goal facts of the form $query(p1(\overline{t1}), p2(\overline{t2}))$; such a fact says the first argument is a subgoal for $p1$, and any answer substitution that solves it should be used to generate answers for $p2(\overline{t2})$ rather than for $p1(\overline{t1})$.

5.3. Projecting Arguments

Query optimization for relational database queries exploits the commutativity of *selection* and *projection* operators with respect to the *join* operator whenever possible, in order to reduce the size of relations that are being joined. This is often referred to as "pushing" selections and projections. Pushing selections is achieved through the use of the Magic Templates transformation; the introduction of recursion makes it necessary to compute auxiliary sets. Pushing projections has also been explored, and the gains can be significant. We will illustrate the idea through examples, and refer the reader to [75] for details.

EXAMPLE 5.2. Consider the transitive closure program of Example 1.1, but with the query $query(Y) :- t(\_, Y)$. The underscore "_" indicates that we do not care about the value in the first argument position; we simply want the set of values that appear in the second argument position of t (with some arbitrary value in the first argument position). Note that such queries are likely to arise during the course of query optimization. □

An adornment that distinguishes don't-care argument positions (d) from the rest (needed or n) was used in [75] to push projections.

EXAMPLE 5.3. Consider a simplified version of the transitive closure program:

$$\begin{aligned} &t(X,Y) &&:- \ t(X,W), e(W,Y). \\ &t(X,Y) &&:- \ e(X,Y). \\ &query(Y) &&:- \ t(\_,Y). \end{aligned}$$

The adorned program — using n and d adornments — is:

$$\begin{array}{lll} t^{dn}(X,Y) & :- & t^{dn}(X,W), e(W,Y). \\ t^{dn}(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t^{dn}(\_,Y). \end{array}$$

It is easy to see that the d argument positions can be uniformly deleted, and this leaves us with a program in which the recursive predicate is unary. □

The previous example illustrated how the adornment algorithm can sometimes push the projection through recursion and thereby reduce the arity of recursive predicates. (The observation that pushing projections could reduce arity of recursive predicates was first made by Aho and Ullman [1], and later Kifer and Lozinskii [46] and Apers et al. [2]. The adornment algorithm in [75] generalizes their results.) The acute reader will have observed that more can be achieved — the recursive rule may be deleted entirely. Algorithms for deleting redundant rules and literals can be utilized to detect this; in fact, since such opportunities are frequently created by pushing projections, we can devise special algorithms for rule and literal deletion that exploit this. The following program is obtained in the above example using the techniques of [75]:

$$\begin{array}{lll} t^{dn}(Y) & :- & e(X,Y). \\ query(Y) & :- & t^{dn}(Y). \end{array}$$

5.4. Counting

Counting ([7; 87; 13]) is a refinement of the Magic Sets approach. Whereas the Magic Sets method restricted the computation to relevant facts, Counting additionally computes indices for each fact that indicate why it is relevant, and this additional information is used to delete some literals from rule bodies and to reduce the arity of some predicates by deleting some argument positions.

Goals correspond to magic facts, as we noted earlier. In the Counting program, magic facts are called *count* facts. The index value for a non-count fact is simply the index value of the goal for which it was generated as an answer. Index values can be encoded in various ways; we will not consider the details.

EXAMPLE 5.4. Consider the program of Example 3.1 again. The Counting algorithm generates:

$$\begin{array}{lll} sg^{bf}(Y,I) & :- & count\_sg^{bf}(X,I), flat(X,Y). \\ sg^{bf}(Y,I) & :- & sg^{bf}(V,I+1), down(V,Y). \\ count\_sg^{bf}(U,I+1) & :- & count\_sg^{bf}(X,I), up(X,U). \\ count\_sg^{bf}(john,0). & & \end{array}$$

We essentially compute (i) "count" facts (analogous to "magic" facts) that are numbered with their distance from the query (the *count* facts), and (ii) values in the free argument positions of program (*sg*) facts (which we will refer to as "answers" in this example), also numbered according to the "count" fact used to generate them. Thus, we only compute sets of goals and answers, all numbered according to their distance from the query. The answers to the original query are the (new, reduced arity) *sg* facts at distance (index value) 0. The key observation here is that we can no longer identify the set of answers to any intermediate subgoals since we only maintain sets of goals and sets of facts, and there may be several goals at a given distance (except 0, where the only goal is the original query). The potential gain in the method derives from this observation — where several goals have the same, or largely overlapping, answer sets we gain by not associating an answer with each of these goals. On the other hand, it is possible to lose in performance if each goal/answer appears at several distances from the query. This can happen for example if there is a cycle in the *up* relation. Indeed, such a cycle can lead to non-termination. It is usually suggested that if a goal is derived with two different distances (i.e. two *count* facts differ only in the index values), the Counting method should be abandoned in favor of Magic Sets. □

5.5. Space Optimization

Bottom-up evaluation derives new facts, but has no mechanism built in to discard facts during the evaluation — all facts are retained till the end of the evaluation. It is important to discard facts during an evaluation, once they are not needed, for otherwise space requirements may grow in an unbounded fashion. Facts are needed to make derivations and to detect duplicate derivations of facts (which may be needed for termination). Consider the following program which computes fibonacci numbers:

$$\begin{aligned}
&R1 : fib(0,1).\\
&R2 : fib(1,1).\\
&R3 : fib(N, X1+X2) :- N>1, fib(N-1, X1), fib(N-2, X2).
\end{aligned}$$

In a Semi-naive evaluation of this program, once a fact $fib(n,\_)$ is deduced, there is no need to keep any facts $fib(m,\_)$ for $m \leq n-2$, since all derivations using such facts have been made, and the facts will not be derived again. Such facts can then be discarded during the evaluation. This issue is studied in detail in [62; 97], where several sufficient conditions for discarding facts are developed.

5.6. Duplicate Elimination

In general, when a fact is derived in a bottom-up evaluation, duplicate checking needs to be done to see if the fact was derived earlier. If it was not, derivations can be made using the fact. If it was, the fact can be discarded. This check is important for termination of many programs. However, for some programs such as the fibonacci program above, with Semi-naive bottom-up evaluation, duplicate facts are never generated. Hence duplicate elimination need not be done for such programs. Maher and Ramakrishnan study this issue in detail in [51], and provide sufficient conditions to detect if a program will not compute duplicate facts.

5.7. Linearizing Programs

An interesting class of program transformations has recently been explored by a number of researchers [39; 113; 88; 77]. The objective is to transform a program that contains non-linear rules into an equivalent one that contains only linear rules; this may make some of the other transformations surveyed in this paper applicable, or permit simplifications in the implementation of the fixpoint evaluation phase. We do not consider these results here due to space limitations.

5.8. Bounded Recursion and Redundant Literals

One interesting optimization that has been studied in some detail is that of *bounded recursion*, where a recursive program is bounded if it can be replaced by a nonrecursive program.

Note that here we are not asking when recursion can be replaced by iteration; rather, we are asking when a logic program containing recursive clauses has an equivalent finite logic program in which no clause is recursive.

EXAMPLE 5.5. The following example is from [61].

$$\begin{aligned} buys(X,Y) &\ :-\ likes(X,Y). \\ buys(X,Y) &\ :-\ trendy(X), buys(Z,Y). \end{aligned}$$

In English, a person X buys a product Y if either X likes Y, or X is trendy and a person Z has bought Y. This recursive program is equivalent to the following nonrecursive program:

$$\begin{aligned} buys(X,Y) &\ :-\ likes(X,Y). \\ buys(X,Y) &\ :-\ trendy(X), likes(Z,Y). \end{aligned}$$

□

In general, detecting bounded recursions is undecidable. A rich classification of sets of programs for which detecting boundedness is decidable or undecidable has been developed in the literature; see, for example, [26; 38; 31; 61; 66; 105]. A related problem is to identify literals and rules that are redundant, i.e., that do not change the set of answers for any input database. For more details about detecting and eliminating redundancy from recursive Datalog programs, see [67].

5.9. Algebraic Properties of Programs

The fixpoint evaluation of a logic program can be refined by taking certain algebraic properties of the program into consideration. Such refinements, and techniques for detecting when they are applicable, have been investigated by several researchers [37; 39; 50; 60; 77]. Ioannidis presents an algebraic formulation of Datalog programs that is particularly suited to reasoning about such properties of programs [39]. The following idea is illustrative of this class of optimizations.

EXAMPLE 5.6. We begin with an example that illustrates *commutativity* of rules. The idea has been studied in the references cited above, and several sufficient conditions are known for detecting when this property holds.

$$\begin{aligned} r_1 : p(X,Y) &\ :- \ a(X,Z), p(Z,Y). \\ r_2 : p(X,Y) &\ :- \ p(X,Z), b(Z,Y). \\ r_3 : p(X,Y) &\ :- \ c(X,Y). \end{aligned}$$

It can be shown that applications of rules r_1 and r_2 commute. That is, $r_1.r_2 = r_2.r_1$. This fact can be utilized to avoid many redundant derivations by evaluating the fixpoint of the program as follows: first apply rule r_3, then apply r_2 as often as possible, and finally apply r_1 as often as possible. □

5.10. Non-Ground Facts

Non-ground facts are useful for several reasons. They are often useful in knowledge representation, where we can say that a fact is true for all values of a variable. They are useful in a database that stores (and possibly manipulates) rules. They are also useful for specifying patterns in queries, and are necessary for representing constraint facts.

However, supporting non-ground facts in a bottom-up evaluation (actually, in general in an evaluation that memos facts) has an associated cost. (1) When making an inference, facts have to have their variables renamed so that they do not share variable names. (2) When a head fact with a large term as an argument is generated, we must share the structure of subterms with facts used to derive it, or do extensive copying. This is easy with ground terms, but with non-ground terms structure sharing becomes more difficult. (3) If variables in a large term get instantiated during the inference, creation of the head fact is made more complicated. (4) Unification of large non-ground terms using currently known techniques is inefficient, especially compared to the cost of unification for ground terms using hash-consing [36; 89]. (5) Indexing of facts containing variables using currently known techniques is inefficient, especially compared to the cost of indexing ground facts using, say, hash tables.

Consider a program to append two lists:

$$\begin{array}{l}
\mathit{append}([], Z, Z). \\
\mathit{append}([X|Y], Z, [X|Y1]) :- \mathit{append}(Y, Z, Y1). \\
\text{Query: ?-}\mathit{append}([A, B, C], [D, E, F], ANS).
\end{array}$$

The Magic Templates rewriting of the above program is

$$\begin{array}{ll}
\mathit{append}([], Z, Z) & :- \mathit{magic\_append}([], Z). \\
\mathit{append}([X|Y], Z, [X|Y1]) & :- \mathit{magic\_append}([X|Y], Z), \\
 & \qquad\qquad \mathit{append}(Y, Z, Y1) \\
\mathit{magic\_append}([A, B, C], [D, E, F]). &
\end{array}$$

If the query uses ground lists, this program runs in $O(n)$ time, where n is the size of the lists to be appended. If the query uses non-ground lists, and the program is evaluated without any special optimizations for non-ground terms, the evaluation takes $O(n^2)$ time.

The structure sharing scheme of Boyer and Moore [14] addresses the first three problems. The idea is to maintain binding environments with terms, and note variable bindings there; this also makes sharing subterms easier. A somewhat complicated naming scheme helps rename all variables in a term in constant time. However, in the worst case, the cost of looking up variable bindings is large since many binding environments may need to be searched. Also, this scheme does nothing to solve the fourth problem, namely the cost of unification.

There appears to be no general solution for theorem proving systems, which is the context in which these problems first appeared. However, in special cases of bottom-up evaluation of logic programs, efficient techniques are possible.

In the case of the append program there is no need to do renaming for the first and last rule of the magic program. For the second rule if the terms to which the variables Y and Z are bound by the first literal are exactly the same (pointer match), there is no need for renaming - the variables in the term to which $Y1$ is bound can be deduced to be the same as those in the terms that Y and Z are bound to. This same optimization can perform unification in constant time. Efficient indexing techniques are possible when Supplementary Magic Templates rewriting is used. No variable instantiation occurs within large terms, and this makes sharing of subterms easy. Putting these optimizations together, the program can be evaluated in $O(n)$ time. A class of programs for which these optimizations are possible is discussed in Sudarshan and Ramakrishnan [96].

6. Extensions to the Horn Clause Logic Programming Paradigm

6.1. Negation

A deductive database query language can be enhanced by permitting negated literals in the bodies of rules in programs. This provides the ability to deal with non-monotonic deduction. However, in the presence of negated literals, a program may not have a least model. For example, the program:

$$p(a) :- \neg p(b).$$

has two minimal models, $\{p(a)\}$ and $\{p(b)\}$, but no least model. The meaning of a program with negation is usually given by some "intended" model ([18; 3; 71; 70; 34; 84; 72; 103], among others).

One important class of negation that has been extensively studied is stratified negation [18; 3]. Intuitively, a program is stratified if there is no recursion through negation. Programs in this class have an intuitive semantics and can also be efficiently evaluated. The following example describes a stratified program.

EXAMPLE 6.1. Consider the following program $P2$:

$$\begin{array}{lll} r1 : anc(X,Y) & :- & par(X,Y). \\ r2 : anc(X,Y) & :- & par(X,Z), anc(Z,Y). \\ r3 : nocyc(X,Y) & :- & anc(X,Y), \neg anc(Y,X). \end{array}$$

Intuitively, this program is stratified because the definition of the predicate *nocyc* depends (negatively) on the definition of *anc*, but the definition of *anc* does not depend on the definition of *nocyc*.

A bottom-up evaluation of $P2$ would proceed SCC-by-SCC. First, we would compute a fixpoint of rules $r1$ and $r2$ (the rules defining *anc*). Rule $r3$ is applied only when the all *anc* facts are known. This can be seen to be as efficient as the evaluation of programs without negation.

However, the Magic Templates transformation of a stratified program may become unstratified, and the evaluation of the resulting unstratified program has received considerable attention in the literature ([4; 44; 42; 10], among others). □

A natural extension of stratified programs is the class of locally stratified programs [71]. Intuitively, a program P is locally stratified if the (propositional) program $Gr(P)$ obtained by taking all ground instances of all rules in P is stratified. Local stratification has been extended to modular stratification in [84]. A program P is said to be modularly stratified if each strongly connected component (SCC) of P is locally stratified after removing instantiated rules containing literals that are false in lower SCCs.

EXAMPLE 6.2. Consider the following program $P3$:

$$\begin{aligned}
&r1 : even(0).\\
&r2 : even(s(X)) :- \neg even(X).
\end{aligned}$$

This program can be seen to be locally stratified. Consider the following variant $P4$ of the above program:

$$\begin{aligned}
&r1 : even(0).\\
&r2 : even(X) :- succ(X,Y), \neg even(Y).\\
&\quad succ(1,0). \quad succ(2,1). \quad succ(3,2).
\end{aligned}$$

Since rule $r2$ can be instantiated to give the same value for X and Y, program $P4$ is not locally stratified. However, it is modularly stratified. Locally and modularly stratified programs can be evaluated using the techniques described in [84; 42]. The evaluation of the Magic Templates transformation of this class of programs has also been considered in the literature ([84; 42; 79]). □

6.2. Set Grouping and Aggregation

The *group-by* construct is used in SQL to partition a relation based on the values of some of the attributes of the relation. A variant of this construct is used in LDL [11; 59] to create nested sets within facts. We use the syntax of LDL in our discussion; however, the semantics we use is from the CORAL[76] system, and is a minor variant of the LDL semantics. The following example illustrates the use of set grouping construct $< >$:

$$multiset\_of\_grades(Class, < Grade >) :- student(Name, Class, Grade).$$

We assume that there is at most one occurrence of the $< >$ operator in the head of any rule, and the operator does not occur in the body. We also assume for now that the program is stratified wrt the $< >$ operator, i.e., the definition of the predicate in the body does not depend on the head predicate if the head of the rule has an occurrence of the $< >$ operator. We first (conceptually) create a multiset of tuples for *multiset_of_grades* using the rule

$$multiset\_of\_grades(Class, Grade) :- student(Name, Class, Grade).$$

Now for each value of $Class$ (in general, each value of the arguments of the head that are not enclosed in the $< >$), we create a multiset containing all the corresponding values for $Grade$. For each value of $Class$ let this multiset be called S_{Class}; we then create a fact $multiset\_of\_grades(Class, S_{Class})$ for each value of $Class$.

Aggregate operations such as $count$, sum, min, and max, which can be applied to sets of values, are permitted in relational database query languages such as SQL. These can be incorporated into logic programming too, using the set grouping construct to create multisets that are aggregated on. Extending the above example, consider the following rule: The following example illustrates the use of aggregation:

$$max\_grade(Class, max < Grade >) :- student(Name, Class, Grade)$$

As before, for each value of $Class$ we create a multiset. But now we apply the aggregate operation max to the multiset, and create a head fact using this value rather than the set itself.

There are several approaches to relaxing the assumption of stratification. Several of the approaches used for negation carry over to set grouping. There are approaches based on weaker forms of stratification such as group stratification and magical stratification [56], or modular stratification [84]. Extensions of the well-founded and stable models to deal with aggregates are considered in [43]. Monotonic programs, where a derivation using an incomplete set does not affect the set of facts computed, are discussed in [25; 56].

In certain contexts, an incomplete set may be used to generate facts that would not be generated using a complete set, without affecting the final answer to some queries on a program. One such context is when subsumption of facts is defined in such a way that facts created using incomplete sets will be found to be subsumed and will be discarded. Ross and Sagiv [86] examine such classes of programs. Cruz and Norvell [27] examine the use of such forms of aggregation in the context of transitive closure. Ganguly, Greco and Zaniolo [33] examine how to evaluate a class of programs that are monotonic in a sense different from that used in [56]. Ganguly et al. [33] and Sudarshan and Ramakrishnan [95] examine rewrite optimizations of programs that use aggregation, and the rewriting can result in unstratified programs. The following program for computing the shortest path length benefits from these optimizations:

$$\begin{array}{lll}
R1: s\_p\_length(X,Y,min(<C>)) & :- & path(X,Y,C). \\
R2: path(X,Y,C1) & :- & path(X,Z,C), \\
 & & \quad edge(Z,Y,EC), \\
 & & \quad C1 = C + EC. \\
R3: path(X,Y,C) & :- & edge(X,Y,C). \\
\text{Query: } ?\text{-}s\_p\_length(X,Y,C). & &
\end{array}$$

For the above program, both the above techniques reduce the number of facts computed from possibly infinite to polynomial, and both evaluation techniques can be implemented to be as efficient as Dijkstra's algorithm for shortest paths.

6.3. Constraint Query Languages

Constraint query languages ([41; 82; 23; 8] among others) differ from deductive database query languages in several ways:

1. With each rule r in a constraint query language, we can have a conjunction of constraints over some algebraic structure, in addition to the other literals in the body of the rule.
2. Each fact $p(\overline{X}; C)$ is a conjunction of constraints over the given structure, a relation is a finite collection of such facts (a disjunction of conjunctions of constraints), and a database is a finite collection of such relations.

Constraint query languages provide a model for reasoning about (potentially) infinite sets of ground facts using finite representations. This is because a constraint fact $p(\overline{X}; C)$ is a finite representation of the (potentially) infinite set of ground facts that satisfy the conjunction of constraints C.

Jaffar and Lassez [40] gave a model theory, a fixpoint theory, and a top-down operational semantics for such programs. Kanellakis et al. [41] gave a bottom-up operational semantics for some classes of constraint query languages.

EXAMPLE 6.3. Consider the following constraint logic program P with a query:

$$\begin{array}{l}
r1: p(X) \;:-\; X \le 5, p1(X), p2(X). \\
r2: p(X) \;:-\; X \ge 7, p2(X). \\
r3: p1(X) :- X \ge 1, X \le 6. \\
r4: p2(X) :- X \ge 3, X \le 9. \\
r5: p2(X) :- X \ge 11, X \le 15. \\
\text{Query: } ?\text{-}p(X), C.
\end{array}$$

where p is the query predicate, and C is a conjunction of constraints on the query. The least model of the program P is the (infinite) set of ground facts $\{p1(a), 1 \leq a \leq 6; p2(b), 3 \leq b \leq 9; p2(c), 11 \leq c \leq 15; p(d), 3 \leq d \leq 5; p(e), 7 \leq e \leq 9; p(f), 11 \leq f \leq 15\}$.

Given the above query, program P can be rewritten using Magic Templates and left-to-right sips for each rule to obtain $\langle P^{mg}, Q^{mg}\rangle$ as follows:

$$\begin{array}{lll}
r1: & p(X) & :- \ magic\_p(X), X \leq 5, p1(X), p2(X). \\
r2: & p(X) & :- \ magic\_p(X), X \geq 7, p2(X). \\
r3: & p1(X) & :- \ magic\_p1(X), X \geq 1, X \leq 6. \\
r4: & p2(X) & :- \ magic\_p2(X), X \geq 3, X \leq 9. \\
r5: & p2(X) & :- \ magic\_p2(X), X \geq 11, X \leq 15. \\
r6: & magic\_p1(X) & :- \ magic\_p(X), X \leq 5. \\
r7: & magic\_p2(X) & :- \ magic\_p(X), X \leq 5, p1(X). \\
r8: & magic\_p2(X) & :- \ magic\_p(X), X \geq 7. \\
Q^{mg}: & magic\_p(X) & :- \ C.
\end{array}$$

We associate the desired query constraints with the corresponding seed magic fact.

The magic program is also a constraint logic program. The magic predicates can be considered to define a relation consisting of the subgoals that a top-down evaluation ([40]) would generate, and the constraints associated with each magic fact correspond to the desired constraints for answers to that subgoal.

We describe the evaluation of the magic program $\langle P^{mg}, Q^{mg}\rangle$ for various constraints C:

- $C \equiv (X \geq 7) \& (X \leq 12)$:
  All $p(a)$ facts, $7 \leq a \leq 9$, and $p(b)$ facts, $11 \leq b \leq 12$ are in the least model of P, and are answers to the query.
  In the bottom-up evaluation of the magic program, $magic\_p(X; X \geq 7, X \leq 12)$ is first derived (using rule Q^{mg}). Rule $r8$ can now be applied to derive $magic\_p2(X; X \geq 7, X \leq 12)$. (Rule $r6$ cannot be applied because the constraints in the body of the rule are not satisfied.) This fact can now be used in rules $r4$ and $r5$ to derive $p2(X; X \geq 7, X \leq 9)$ and $p2(X; X \geq 11, X \leq 12)$. These $p2$ facts can be used in the body of rule $r2$ to derive $p(X; X \geq 7, X \leq 9)$ and $p(X; X \geq 11, X \leq 12)$.
  This gives a finite representation of the desired answers to the query, and mimics the top-down evaluation as well. Note that the bottom-up evaluation does not compute any facts for $p1$ since these are irrelevant to answering the query.
- $C \equiv X \leq 2$:
  There are no p facts in the least model that are answers to the query. In the bottom-up evaluation of the magic program, $magic\_p(X; X \leq 2)$ is first derived. Rule $r6$ can now be applied to derive $magic\_p1(X; X \leq 2)$. (Rule $r8$ is not applicable.) This fact can be used in the body of rule $r3$

to derive $p1(X; X \geq 1, X \leq 2)$. Rule $r7$ can now be applied to derive $magic\_p2(X; X \geq 1, X \leq 2)$. However, this fact cannot be used in either rule $r4$ or rule $r5$ and the evaluation terminates.
Again, this mimics the top-down evaluation. □

6.4. Quantitative Logic Programs

Quantitative logic programs (also referred to as fuzzy databases) ([93; 101; 45], among others) differ from ordinary logic programs in several ways:

1. With each fact $p(\overline{a})$, we associate a "certainty," $0 \leq \alpha \leq 1$, which is a measure of the strength of the evidence for $p(\overline{a})$.
2. With each rule r in a quantitative logic program, we associate a certainty function f_r that is used to compute the certainty of the head fact, given the certainties of facts in the body.
3. With each predicate p, we associate an evidence combination function F_p that combines the strengths of evidences obtained from different sources all supporting the same fact.

This provides a model for rule-based reasoning in expert systems, where the usual logical reasoning in terms of the truth values **true** and **false** is insufficient.

We describe the restricted case of K-standard sequence logics ([93; 101]), and refer the interested reader to Kifer and Li [45] for more general types of quantitative logic programs. Quantitative logic programs based on K-standard sequence logics were considered by Van Emden [101], who gave a model theory, a fixpoint theory, and a top-down operational semantics for such programs. A rule in a quantitative logic program with K-standard sequence logic is of the form:

$$r : p(\overline{X}) : c_r * \min\{\alpha_1, \alpha_2, \ldots, \alpha_n\} :- p1(\overline{X_1}) : \alpha_1, \ldots, p_n(\overline{X_n}) : \alpha_n.$$

Thus, the rule certainty function, f_r, associated with rule r is

$$f_r = c_r * \min\{\alpha_1, \ldots, \alpha_n\}$$

where c_r is some constant, $0 < c_r \leq 1$, associated with the rule. The evidence combination function $F_p(\{\alpha_1, \ldots, \alpha_m\})$ in such a logic is defined as $\max\{\alpha_1, \ldots, \alpha_m\}$.

EXAMPLE 6.4. Consider the following program $P1$ in K-standard sequence logic, with query Q:

$r1 : p(X) : 0.5 * \min\{\alpha_1, \alpha_2\} :- p1(X) : \alpha_1, p2(X) : \alpha_2.$
$r2 : p(X) : 0.45 * \alpha_1 \qquad :- p2(X) : \alpha_1.$
$r3 : p1(1) : 0.4.$
$r4 : p2(1) : 0.5.$
$r5 : p2(2) : 0.25.$
Query: ?-$p(1) : c_p$.

where p is the query predicate, and c_p is the desired certainty for $p(1)$. The least model of the program $P1$ is $\{p1(1) : 0.4, p2(1) : 0.5, p2(2) : 0.25, p(1) : 0.225, p(2) : 0.1125\}$.

Given the above query, program $P1$ can be rewritten using (a variant of) Magic Templates and left-to-right sips for each rule to obtain $\langle P1^{mg}, Q^{mg}\rangle$ as follows:

$$\begin{array}{lll}
r1: & p(X) : 0.5 * \min\{\alpha_1, \alpha_2\} & :- \ \mathit{magic\_p}(X) : \alpha_0, p1(X) : \alpha_1, \\
 & & \quad p2(X) : \alpha_2, \\
 & & \quad \alpha_0 \leq 0.5 * \min\{\alpha_1, \alpha_2\}. \\
r2: & p(X) : 0.45 * \alpha_1 & :- \ \mathit{magic\_p}(X) : \alpha_0, p2(X) : \alpha_1, \\
 & & \quad \alpha_0 \leq 0.45 * \alpha_1. \\
r3: & p1(1) : 0.4 & :- \ \mathit{magic\_p}1(1) : \alpha_0, \alpha_0 \leq 0.4. \\
r4: & p2(1) : 0.5 & :- \ \mathit{magic\_p}2(1) : \alpha_0, \alpha_0 \leq 0.5. \\
r5: & p2(2) : 0.25 & :- \ \mathit{magic\_p}2(2) : \alpha_0, \alpha_0 \leq 0.25. \\
r6: & \mathit{magic\_p}1(X) : \alpha_0/0.5 & :- \ \mathit{magic\_p}(X) : \alpha_0. \\
r7: & \mathit{magic\_p}2(X) : \alpha_0/0.5 & :- \ \mathit{magic\_p}(X) : \alpha_0, p1(X) : \alpha_1, \\
 & & \quad \alpha_1 \geq \alpha_0/0.5 \\
r8: & \mathit{magic\_p}2(X) : \alpha_0/0.45 & :- \ \mathit{magic\_p}(X) : \alpha_0. \\
Q^{mg} & : \mathit{magic\_p}(1) : c_p. &
\end{array}$$

We associate the desired query certainty with the corresponding seed magic fact $\mathit{magic\_p}(1)$.

Note that the magic program cannot be viewed as a quantitative logic program in K-standard sequence logic. The magic predicates can be considered to define a relation consisting of the subgoals that a top-down evaluation would generate, and the number associated with the magic fact corresponds to the certainty desired for answers to that subgoal—this number can thus be semantically viewed as a "cutoff."

We describe the evaluation of the magic program $\langle P1^{mg}, Q^{mg}\rangle$ for various values of c_p:

– $c_p = 0.2$:

Since $p(1) : 0.225$ is in the least model of $P1$, so is $p(1) : 0.2$. The desired answer to the query is **yes**.

In the bottom-up evaluation of the magic program, $\mathit{magic\_p}(1)$ is first derived (using rule Q^{mg}) with a cutoff of 0.2. Rule $r6$ can now be applied to derive $\mathit{magic\_p}1(1)$ with a cutoff of 0.4, and rule $r8$ can be applied to derive $\mathit{magic\_p}2(1)$ with a cutoff of 0.444. Rule $r3$ can then be applied to derive $p1(1)$ with certainty 0.4 (since the constraint in the body is satisfied); rule $r4$ can also be applied to derive $p2(1)$ with certainty 0.5. Now, the magic fact $\mathit{magic\_p}2(1)$ also can be derived using rule $r7$ with a cutoff of 0.4. The fact $p2(1) : 0.5$ is now derived again. Finally, rule $r1$ can be applied to compute $p(1) : 0.2$, and rule $r2$ can be applied to compute $p(1) : 0.225$.

This gives the required answer to the query, and mimics the top-down evaluation ([101]) as well. Note that the bottom-up evaluation does not compute facts $p2(2) : 0.25$ and $p(2) : 0.1125$ which are irrelevant to answering the query.

– $c_p = 0.25$:

The desired answer to the query is **no**. In the bottom-up evaluation of the magic program, *magic_p*(1) is first derived with a cutoff of 0.25. The magic fact *magic_p*1(1) can now be derived (using rule $r6$) with a cutoff of 0.5 (indicating that we need to compute $p1$ with a certainty of at least 0.5); *magic_p*2(1) can also be derived (using rule $r8$) with a cutoff of 0.556. However, neither $p1(1)$ nor $p2(1)$ can be derived, since the constraints in the appropriate bodies (of rules $r3$ and $r4$) fail. Since no $p1$ or $p2$ fact is derived, no further derivations are made, and there is no answer to the query (as expected).

Again, this mimics the top-down evaluation, and no irrelevant facts are computed. □

7. Deductive Database System Implementations

There have been a number of implementations of deductive databases. In this section we briefly discuss some of them.

One of the early implementations was LDL [59; 20; 21]. LDL is a main-memory database system implemented at MCC Austin. It extends the declarative semantics of Horn clauses by allowing set grouping and aggregate operations. The treatment of sets allows set-terms to be specified in the body of rules; set-matching is used to support this feature. It also allows rules to perform updates, which means that such rules do not have the usual declarative semantics. Rather, their semantics is defined in a constructive fashion [58]. LDL also provides some control primitives such as choice, provides a module feature for organizing code, and provides primitives for integration with C.

NAIL! [98; 53; 52] was another early deductive database system. The current version is compiled into a set-oriented procedural language called Glue [69]. NAIL! provides higher order syntactic features, using the semantics of Hilog [19]. Glue/NAIL! is currently implemented on top of Prolog, and is main-memory oriented.

EKS-V1 [107] is a deductive database system that combines features of disk-oriented and in-memory databases. It is implemented on top of the MEGALOG logic programming system which supports access to secondary storage. It provides facilities for handling negation and aggregation in a set-oriented fashion. Unlike the other deductive database systems discussed here, it provides support for integrity constraint maintenance and hypothetical reasoning.

Aditi [100] is a deductive database system that is primarily disk oriented. It provides support for parallelization of the low level operations such as joins. It is integrated with Prolog, and is compiled into an intermediate language which is then interpreted. All relational operations are performed with relations assumed to be disk resident, and join techniques such as sort-merge and hash-join are used, which means that Aditi is aimed at applications that are I/O intensive, but typically perform few iterations.

Starburst SQL (see e.g. [56]) extends SQL with recursive view definitions. This provides greater expressive power to SQL users, while providing the I/O optimization facilities of SQL in a direct manner for recursive applications.

CORAL [76] is a deductive database system developed by the authors, and is described in detail in a separate section. LDL, NAIL! and CORAL belong in one family of deductive database systems in that they are main-memory oriented. They attempt to provide the power of logic programming using complex terms built using uninterpreted function symbols, and add new constructs to standard Horn clause logic programs. EKS-V1, Aditi and Starburst SQL on the other hand are oriented more towards secondary storage, and provide less support for complex terms.

8. CORAL—A Deductive Database Programming Language

CORAL[3] is a database programming language being developed at UW-Madison. An important goal of CORAL is to provide a powerful declarative query language. A second goal of CORAL is to provide a clean interface between an imperative language (C++) and the declarative language. This is necessary for several reasons such as permitting the user to perform actions such as updates and to implement time critical functions more efficiently. To this end C++ is extended with several types, constructs and library functions.

A CORAL program can be viewed as a collection of *declarative modules*, and an *imperative module*. CORAL provides a convenient syntax for defining predicates using the extended version of C++, and for invoking queries on the declarative modules from the imperative module. Declarative modules are compiled into C++ classes, and the runtime system is written in C++. Thus, CORAL reduces the impedance mismatch between the declarative modules and the C++ module at both the language level and the implementation level.

While there are many similarities to the other deductive database systems mentioned earlier, there are important differences as well. For example, CORAL supports non-ground terms and ground multisets. The use of general updates is not allowed in declarative rules. A number of the control

[3] CORAL stands for "COmbining Relations And Logic".

features supported in CORAL are novel, as is the approach to integrating with an imperative language such as C++.

8.1. Declarative Modules

Declarative programming is supported by allowing the user to organize sets of rules and facts into *declarative modules*. We introduce the syntax using a program to append two lists.

module *Listroutines*.
export *append* (bbf, bfb, fbb).

$append([\,], L, L)$.
$append([H \mid T], L, [H \mid L1]) :- append(T, L, L1)$.

end_module.

This program illustrates the notion of modules, and CORAL's support for complex objects such as lists. The *export* statements at the beginning of each module define what forms of external queries are permitted on the module ("b" denotes an argument that must be bound in the query, and "f" an argument that can be free).

CORAL allows facts to have variables within them. Thus, it is possible to query *append* as follows:

Query: ?-$append([1,2,3,4,X],[Y,Z],ANS)$.

and get the answer

$ANS = [1,2,3,4,X,Y,Z]$

In this CORAL differs from Aditi, EKS-V1, LDL, Glue/NAIL! and Starburst SQL, which restrict the facts in a database to be ground. The cost and benefits of non-ground facts is discussed in Section 5.10.

8.1.1. Sets and Negation

The keyword **not** is used as a prefix to indicate a negated body literal. A program is non-floundering if all variables in a negated literal are ground before the literal is evaluated (in the left-to-right rule order). CORAL supports a class of programs with negation that properly contains the class of non-floundering modularly stratified programs (Ross [84]).

Sets can be constructed by *set-enumeration*, or by *set-grouping*. Set grouping was discussed in Section 6.2. The following rule (fact) illustrates set-enumeration.

$children(john, \{mary, peter, paul\})$.

All elements of sets generated thus must be ground terms (i.e., not contain any variables). General matching or unification of sets (where one or both of the sets can have variables, respectively) is not supported. CORAL requires the use of the set-grouping operator to be modularly-stratified (in the same way as negation). When programs allow the creation of sets (and nested structures that include sets) the domain (or universe) of discourse has to be extended beyond the Herbrand universe that is standard in logic programming. Beeri et al. [12] describe such an extended Herbrand universe.

CORAL provides several standard operations on sets and multisets as built-in predicates. These include **member**, **union**, **intersection**, **difference**, **multisetunion**, **cardinality**, **subset**, and **set**. CORAL also provides several aggregate operations for use on sets and multisets such as **count**, **min**, **max**, **sum**, **product**, **average** and **any**.

8.2. Other Declarative Module Features

CORAL uses bottom-up evaluation as the basic paradigm for evaluation of declarative modules. It uses a variety of optimization techniques including program transformations such as Magic Templates ([13; 73]), optimizations of bottom-up evaluation such as Semi-naive evaluation, rule and predicate orderings, efficient unification techniques such as hash-consing [36; 89] and sharing of structure for ground terms, and efficient indexing techniques. Currently, evaluation is completely in-memory, although we plan to implement disk-resident relations in future versions of CORAL. More optimization techniques will be incorporated in future versions of CORAL.

Modules are the unit of compilation in CORAL. Numerous annotations are allowed in modules, and many of these have a global effect on the module. Unlike Glue/NAIL!, where modules have only a compile time meaning and no run-time meaning, modules in CORAL have important run-time semantics. Several run-time optimizations are done at the module level.

In most cases, facts (other than answers to the query) computed during the evaluation of a module are best discarded to save space (since bottom-up evaluation stores many facts, space is generally at a premium). Module calls provide a convenient unit for discarding intermediate answers, which may take up too much space otherwise. By default CORAL does precisely this — it discards all intermediate facts and subgoals computed by a module at the end of a call to the module. This can be overridden by the user, by using an annotation **save_module** within a module definition.

CORAL provides utilities to take a text file organized as a table, parse it into fields and records, and convert it into a relation. Similarly, utilities for output of relations in tabular form are also provided. This provides some of the features of the AWK string manipulation language that is available on Unix systems.

CORAL supports Ordered Search [79], which is an evaluation mechanism that orders the use of generated subgoals in a program. Subgoals and answers to subgoals are generated asynchronously, as in the regular bottom-up evaluation of the magic program. However, the order in which generated subgoals are made available for use is somewhat similar to a top-down evaluation. Ordered search maintains information about dependencies between subgoals, which can be used to evaluate a large class of programs with negation. It also provides an ordering to the computation that hides subgoals; this could be useful in situations when only a single answer is desired for a query.

CORAL provides some basic facilities for debugging of programs. A trace facility lets the user know what rules are being evaluated, and prints out answers and subgoals as they are generated. CORAL also provides some high-level profiling facilities, that use as the unit of measurement the unification operation (unification of large subterms are treated as generating recursive calls to unification for the purpose of this count).

8.3. Imperative Modules in CORAL

An imperative module is a program in an imperative language, which consists of C++ augmented by adding a layer of new types and constructs. We introduce some features of imperative modules using the program in Figure 1, which updates the salary of a person depending on the number of employees that work for the person (directly or indirectly). The program gives an intuitive idea of the features we provide for imperative modules.

A *relation* is a set of tuples. The user can create two types of relations: *unindexed relations* and *indexed relations*. Indices can be added to an existing *indexed* relation by means of a procedure call.

A C++ user can also directly access a database relation (not just get a copy of it) by providing the name of the relation and its arity. CORAL provides facilities to insert tuples into, and delete tuples from relations using the "+ =" and "− =" operators. A procedure update_tuple to update tuples in a relation is also provided, as illustrated in the example.

CORAL provides two iterative constructs for accessing the tuples of a relation (the tuples are returned in an arbitrary order). Both these are illustrated in the example. FOR_EACH_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument. FOR_EACH_MATCHING_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument that matches the pattern specified by the third argument. A variety of other functions are available to the imperative language programmer to manipulate relations. These include all the set and aggregate functions described earlier.

A C++ user can invoke a query on a relation that is defined declaratively

```
void Update_Sals (Relation *emp)
{

    TupleType(EmpTuple,3)
        TupleArg(1, string, ename);   TupleArg(2, string, mname);
        TupleArg(3, int, sal);
    EndTupleType(EmpTuple)
    TupleType(NumempsTuple,2)
        TupleArg(1, string, mname);   TupleArg(2, int, numemps);
    EndTupleType(NumempsTuple)
    EmpTuple *emp_tuple, new_tuple;
    NumempsTuple query, *result_tuple, pattern;
    Relation *numofemps = new IndexedRelation(2);

    query.set_mname(make_var(0));
    query.set_numemps(make_var(1));
    call_coral ("numofemps", &query, numofemps);

    FOR_EACH_TUPLE (emp_tuple, emp) {
        pattern.set_mname( emp_tuple→ename());
        pattern.set_numemps( make_var(0) );
        int newsal = emp_tuple→sal();
        FOR_EACH_MATCHING_TUPLE(result_tuple, numofemps,
                                              &pattern) {
            newsal += 10* result_tuple→numemps();
        } END_EACH_TUPLE(result_tuple)
        copy_tuple(emp_tuple, new_tuple);
        new_tuple→set_sal(newsal);
        update_tuple(emp, emp_tuple, new_tuple);
    } END_EACH_TUPLE(emp_tuple)

}
```

module Employee.
export *numofemps* (bf).

$$\begin{array}{ll} worksfor(E,M) & :-\ emp(E,M,S). \\ worksfor(E,M) & :-\ worksfor(E,E1), \\ & \quad\ emp(E1,M,S) \\ numofemps(M,\text{count}(set(<E>))) & :-\ worksfor(E,M). \end{array}$$

end module.

Fig. 1. Updating Employee Salaries

(and exported by a declarative module), using the procedure call_coral illustrated in the example. There are two variants of this procedure, one of which takes a single query, and the other a set of queries. In later versions of CORAL we plan to allow inline declaration of a declarative module within imperative code. This will provide a simpler syntax for calling declarative modules and could be interpreted as a direct extension of the C++ language. CORAL provides a simple convention for defining predicates using C++ code.

To provide efficient support for novel applications, CORAL allows the user to create new abstract data types and integrate them with the declarative query language in a simple and clean fashion, without modifying or recompiling system code. For instance, when dealing with DNA sequences, a type *sequence* that provides several built-in operations such as approximate subsequence matching, indexing etc. is very useful.

9. Conclusion

We have reviewed a collection of results on bottom-up evaluation of logic programs, and attempted to place them in the perspective of a coherent approach to logic program evaluation. The main points, in our opinion, are the following:

– Efficient bottom-up evaluation methods are available that are sound and complete with respect to the declarative semantics.
– Systems based upon these methods are being developed, and offer rich support for rule-based applications.

Acknowledgements

This paper is based upon a tutorial to be presented at COMPEURO 92 by R. Ramakrishnan. It has borrowed some material from other papers, in particular [63].

This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

References

1. Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, 1979.
2. P.M.G. Apers, M.A.W. Houtsma, and F. Brandse. Processing recursive queries in relational algebra. In *Data and Knowledge (DS-2), Proc. of the Second IFIP 2.6 Working Conference on Database Semantics*, pages 17–39. North Holland, 1986.

3. K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, San Mateo, Calif., 1988.
4. I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. Journal of Logic Programming. To Appear.
5. I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.
6. Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems.* Springer-Verlag, 1985.
7. Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.
8. Marianne Baudinet, Marc Niezette, and Pierre Wolper. On the representation of infinite temporal data and queries. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 280–290, Denver, Colorado, May 1991.
9. R. Bayer. Query evaluation and recursion in deductive database systems. Unpublished Memorandum, 1985.
10. C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Valid computations and the Magic implementation of stratified programs. Manuscript, September 91.
11. Catriel Beeri, Shamim Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, California, March 1987.
12. Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *The Journal of Logic Programming*, pages 181–232, 1991.
13. Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.
14. R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Computational Logic*, pages 101–116, 1972.
15. F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Procs. International Conference on Extending Database Technology*, February 1988.
16. Francois Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.
17. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.
18. Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, April 1985.
19. Weidong Chen, Michael Kifer, and Davis S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, 1989.
20. D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the International Conference on Very Large Databases*, pages 195–204, 1989.
21. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, 1990.
22. L. Cholvy and R. Demelombe. Querying a rule base. In *Procs. 1st International*

Conference on Expert Database Systems, pages 365–371, April 1986.
23. Jan Chomicki. Polynomial time query processing in temporal deductive databases. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 379–391, Nashville, Tennessee, April 1990.
24. S.R. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 200–216, Philadelphia, Pennsylvania, March 1989.
25. M. P. Consens and A. O. Mendelzon. Low complexity aggregation in Graphlog and Datalog. In *Procs. International Symposium on Database Theory*, Paris, 1990.
26. Stavros S. Cosmadakis, Haim Gaifman, Paris Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the Twentieth Symposium on the Theory of Computation*, pages 477–490, Chicago, Illinois, May 1988.
27. I. F. Cruz and T. S. Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th Int'l Conf. Data Engineering*, pages 384–389, 1989.
28. Suzanne W. Dietrich and David S. Warren. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987.
29. G. Dong. On distributed processing of Datalog queries by decomposing databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–35, Portland, Oregon, June 1986.
30. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19(1):17–37, 1982.
31. Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 106–115, Ithaca, New York, June 1987.
32. S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of Datalog queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990. To appear.
33. Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.
34. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.
35. R Gonzalez-Rubio, J. Rohmer, and A. Bradier. An overview of DDC: Delta Driven Computer. In *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, Lecture Notes in Computer Science, No. 258, pages 414–433, 1987.
36. E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan, May 1974.
37. A. Richard Helm. Detecting and eliminating redundant derivations in deductive database systems. Technical Report RC 14244 (#63767), IBM Thomas Watson Research Center, December 1988.
38. Yannis E. Ioannidis. Bounded recursion in deductive databases. *Algorithmica*, 1(4):361–385, October 1986.
39. Yannis E. Ioannidis and Eugene Wong. Towards an algebraic theory of recursion. Technical Report 801, Computer Sciences Department, University of Wisconsin-Madison, October 1988.
40. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL*, pages 111–119, Munich, January 1987.
41. Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.
42. David Kemp, Divesh Srivastava, and Peter Stuckey. Magic sets and bottom-up

evaluation of well-founded models. In *Proceedings of the International Logic Programming Symposium*, pages 337–351, San Diego, CA, U.S.A., October 1991.

43. David Kemp and Peter Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., October 1991.
44. J.M. Kerisit and J.M. Pugin. Efficient query answering on stratified databases. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 719–725, Tokyo, Japan, November 1988.
45. Michael Kifer and Ai Li. On the semantics of rule based expert systems with uncertainty. In *Proceedings of 2nd ICDT*, Bruges, Belgium, 1988.
46. Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive databases. In *Proceedings of the Advanced Database Symposium*, Tokyo, Japan, 1986.
47. Michael Kifer and Eliezer L. Lozinskii. SYGRAF: Implementing logic programs in a database style. *IEEE Transactions on Software Engineering*, 1988.
48. J. J. King. QUIST: A system for semantic query optimization in relational databases. In *Proceedings of the International Conference on Very Large Databases*, pages 510–517, August 1981.
49. J. Kuittinen, O. Nurmi, S. Sippu, and E. Soisalon-Soininen. Efficient implementation of loops in bottom-up evaluation of logic queries. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
50. Michael J. Maher. *Semantics of Logic Programs.* PhD thesis, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1985.
51. Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990.
52. Katherine Morris, Jeffrey F. Naughton, Yatin Saraiya, Jeffrey D. Ullman, and Allen Van Gelder. YAWN! (Yet Another Window on NAIL!). *Database Engineering*, December 1987.
53. Katherine Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming*, 1986.
54. A. Motro. SEAVE: The mechanism for verifying user presuppositions. *ACM TOIS*, 4(4):312–330, October 1986.
55. I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990.
56. Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
57. Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 314–330, Nashville, Tennessee, April 1990.
58. S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1988.
59. Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.
60. Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.
61. Jeffrey F. Naughton. Data independent recursion in deductive databases. *Journal of Computer and System Sciences*, 38(2):259–289, April 1989.
62. Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without repeating it. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
63. Jeffrey F. Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic pro-

grams. In J-L. Lassez, editor, *Computational Logic: Essays in Honor of Alan Robinson*. The MIT Press, 1991.
64. Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.
65. Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–242, Portland, Oregon, May 1989.
66. Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 227–236, San Diego, California, March 1987.
67. Jeffrey F. Naughton and Yehoshua Sagiv. Minimizing expansions of recursions. In Hasan Ait-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 1, pages 321–349, San Diego, California, 1989. Academic Press, Inc.
68. F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the twenty-first Annual Meeting of the Association for Computational Linguistics*, 1983.
69. Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 308–317, 1991.
70. H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, 1988.
71. T.C. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.
72. T.C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Seventh International Logic Programming Conference*, pages 459–477, 1990.
73. Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.
74. Raghu Ramakrishnan. Parallelism in logic programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990.
75. Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential Datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.
76. Raghu Ramakrishnan, Per Bothner, Divesh Srivastava, and S. Sudarshan. Coral: A database programming language. In Jan Chomicki, editor, *Proceedings of the NACLP '90 Workshop on Deductive Databases*, October 1990. Available as Report TR-CS-90-14, Department of Computing and Information Sciences, Kansas State University.
77. Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Vardi. Prooftree transformation theorems and their applications. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, March 1989.
78. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
79. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. Manuscript, submitted for publication, 1991.
80. Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

81. Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In preparation (full version of paper that appeared in ILPS'91), 1992.

82. Peter Z. Revesz. A closed form for datalog queries with integer order. In *International Conference on Database Theory*, pages 187–201, France, December 1990.

83. J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

84. Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

85. Kenneth Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

86. Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the post-ILPS'91 Workshop on Deductive Databases*, 1991.

87. Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

88. Yatin Saraiya. Linearizing nonlinear recursions in polynomial time. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 182–189, Philadelphia, Pennsylvania, March 1989.

89. M. Sassa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(4):31–34, June 1976.

90. Helmut Schmidt. *Meta-Level Control for Deductive Database Systems.* Lecture Notes in Computer Science, Number 479. Springer-Verlag, 1991.

91. H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

92. T. Sellis, C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 404–412, June 1988.

93. Ehud Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proceedings of IJCAI*, pages 529–532, 1983.

94. Seppo Sippu and Eljas Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

95. S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, September 1991.

96. S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In preparation, 1992.

97. S. Sudarshan, Divesh Srivastava, Raghu Ramakrishnan, and Jeff Naughton. Space optimization in the bottom-up evaluation of logic programs. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, 1991.

98. Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(4):289–321, September 1985.

99. Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

100. Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, and Peter Stuckey. The Aditi deductive database system. In *Proceedings of the NACLP'90 Workshop on Deductive Database Systems*, 1990.

101. M. H. Van Emden. Quantitative deduction and its fipoint theory. *Journal of Logic Programming*, (1):37–53, 1986.

102. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

103. A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded seman-

tics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

104. Allen Van Gelder. A message passing framework for logical query evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 349–362, Washington, DC, May 1986.

105. Moshe Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 341–351, Austin, Texas, March 1988.

106. L. Vieille, P. Bayer, and V. Küchenhoff. Integrity checking and materialized views handling by update propagation in the EKS-V1 system. Technical report, CERMICS - Ecole Nationale Des Ponts Et Chaussees, France, June 1991. Papport de Recherche, CERMICS 91.1.

107. L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, 1990.

108. Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, South Carolina, 1986.

109. Laurent Vieille. Database complete proof procedures based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, 1987.

110. Laurent Vieille. From QSQ towards QoSaQ: Global optimizations of recursive queries. In *Proc. 2nd International Conference on Expert Database Systems*, April 1988.

111. J. Widom, R.J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the International Conference on Very Large Databases*, pages 275–285, August 1991.

112. O. Wolfson and A. Silberschatz. Sharing the load of logic program evaluations. In *Proceedings of the 7th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, March 1989.

113. W. Zhang, C. T. Yu, and D. Troy. A necessary and sufficient condition to linearize doubly recursive programs in logic databases. Unpublished manuscript, Department of EECS, University of Illinois at Chicago, 1988.

PROVING CORRECTNESS OF EXECUTABLE PROGRAMS

KIT LESTER
C.Eng., School of Information Science
Portsmouth Polytechnic
Southsea PO4 8JF, England.

February 14, 1992

Abstract. We claim to be "Software Engineers", yet our methods of software construction are mainly intuitive, and generally have an air of "string and chewing gum" construction. If we are truly to *Engineer* programs, we instead need mathematically-based methods either of constructing the programs, or of verifying intuitively-constructed programs.[1] Most previous papers on such mathematical methods have dealt only with the correctness of program source code; some have dealt additionally with program specification; unusually, this paper further takes the discussion all the way to the executable program.[2]

1. Introduction

A typical definition of "Engineering" in general is

> "The effective manipulation of our environment to our advantage using methods based on mathematical understanding of the relevant aspects of the environment."

There are two main thrusts to that definition:
– "effective"; and
– "mathematical understanding".

For most engineering disciplines, *effective* is usually interpreted in terms of having standard design methods and construction processes, together with understandings of relevant economic, logistic, legal, contractual and ethical issues. In the software-construction domain, we have made some progress in these aspects of "effectiveness".

By contrast, our *mathematical understanding* of how to construct software is limited, and is seldom capitalized upon in real projects. So the purpose of this paper

[1] This is no new observation: at the inaugural conference of EDSAC on 24th June 1949, Alan Turing said:

> "It is of course important that some efforts be made to verify the correctness of the assertions that are made about a routine. There are essentially two types of method available, the theoretical and the experimental. In the extreme form of the theoretical method, a watertight mathematical proof is provided for the assertion. In the extreme variety of the experimental method, the routine is tried out on the machine with a variety of initial conditions."

But in the "computer explosion" of the 1960's, that clarity of vision was lost, and we almost exclusively resorted to Turing's *extreme* variety of experimental method. The ideas didn't die in the '60s — most subsequent work seems to trace back ultimately to Floyd 67 — but were largely ignored, then and since.

[2] If we want a 100% reliable *system*, we will also need proven *hardware*. This places a higher level of demand on the Electronic Engineers than they normally meet for more than small building-blocks of a CPU.

P. Dewilde and J. Vandewalle (eds.), Computer Systems and Software Engineering, 325–353.

is to overview[3] what we have or would desire throughout the software construction process, all the way to the executable machine code.

Specifically, the paper discusses:

– whether and what form of "mathematical understanding" we need;
– issues in specification;
– issues in source code construction;
– issues in translation to machine code.

The last of those has not been widely discussed before, so (more than the other sub-topics) it's given extended consideration throughout the paper, from title to references.

2. Basic Issues

2.1. But do we need "Mathematical Understanding" in Software Construction?

It's painfully evident to anyone who regularly reads a computing trade newspaper that there are unacceptably-frequent news items concerning software that has gone technically "wrong" in up to three main ways:[4]

– the system behaves differently than required in its specification;
– the system is slower than required by its specification, or lacks some capacity required by its specification; or
– the specification doesn't correspond to what the user needed...

And for each fiasco that gets publicised, how many do the trade press not hear about?

So clearly there's a need for better methods than are normally used!

The question then is "what sort of better method?"

Since the "Software Crisis" period around 1970 there have been a number of *minor* technical improvements to our intuition-based methods. The most significant of those improvements is that we have begun to construct software in a modular fashion[5], with the individual module small enough to be (we hope) understandable,

[3] This requires the paper to be tutorial in nature, with various consequences such as occasional "folksy" style, heavy dependence on footnotes to sidetrack material distracting at a first reading, incremental development of some of the more complex concepts, etc.

[4] Projects also often go wrong in *non-technical* ways: e.g. delivered late or incomplete; over budget, etc. In one recent case, a contractor delivered software for a CPU that had, during the development, ceased to be sold: the purchaser hadn't realized, and the contractor "hadn't appreciated the need to draw it to the purchaser's attention".

[5] Although the word "module" is recent, mathematicians have been using recognizably modular methods since at least Descartes: mechanical engineers have been bolting separately-designed units together since the development of the British Standard Whitworth thread: and electronics engineers have been plugging independently-designed and constructed circuit boards into backplanes since at least the early 1930's. Modularization is hardly a novel Engineering principle.

(There's a double link to Computing history here: Joseph Whitworth devised his thread for nuts and bolts as part of his work as "Chief Mechanic" to Charles Babbage's attempt to build a "Universal Calculating Engine" in the 1830's. And Babbage's programmer, Ada Augusta Lovelace, even foresaw and therefore *more probably* correct — but with what statistical confidence level on that "probably"? To fully capitalize on that modularization we need to use a language that helps enforce modularization (e.g. Ada, Eiffel and certain others of its object-oriented brethren, Modula, Extended Pascal), but these languages are even now used for only a minority of applications — most programs are still built using 1960's technology...

At best, these improvements mitigate the problems: at worst, we neglect to use the improvements. The only way in which the "Software Crisis" has passed is that we've got used to the problems.

Yet, since 1970, we have built more and more complex computer systems. In the 1980's this has been compounded by an increasing proportion of these complex systems being in some way "critical"[6], meaning that faults in their design or construction would lead to utterly unacceptable consequences ranging from death and injury to major financial losses. For these systems, we need that they *be* correct, not "probably" correct![7]

So we need much more powerful improvements in our methods than those of the last two decades.

Whenever any of the traditional Engineering disciplines have come to the point of needing "better methods", the better methods that evolved were mathematically-based. Likewise, in software construction mathematics-based methods have been proposed, and (like the other disciplines) we see no other kinds of better basis. We've even developed some mathematically-based methods to the point where they are usable for substantial classes of program, now, at some cost. To a large extent the cost appears to be that we lack economies of scale — e.g. first use of such methods in a company will mean a heavy training bill. (The notations themselves can be learnt as quickly as a programming language: it's the associated methods which take time to acquire. But there are possible cost savings as well: see later.)

If we have a problem in developing these new methods, it's that we see too many detailed possibilities. Quite apart from the resulting inefficiencies of lacking an obvious Standard method, the differences have been a distraction, with much energy spent on trying to discover which variant is "best" when currently we probably only have preliminary proposals. To date, most of the "mathematical methods" proposed for software "correctness" have been mainly oriented towards ensuring behaviour according to specification, but we'll see that some have spin-

the need to break programs down into manageable parts. Ada Augusta's contribution especially makes the pre-1970 non-modularization seem a neglect, rather than later use of modularization being a success.)

[6] E.g. life-critical systems such as fly-by-wire planes (including the Boeing "400 series" and the Airbus 320 — i.e. passenger "jumbos"); medical life-support systems; control systems for petrol refineries; financially-critical systems such as major banking and insurance systems; etc.

[7] For bigger programs, this "probably" almost means "probably not": suppose we could put a statistical measure of confidence on pieces of program, and had 99.9% confidence in *each* module of a 1000-module system — i.e. for each module, a .999 probability that it is error-free — then the probability of the whole system being error-free will be .999 raised to the power 1000, which is .3677 — i.e. almost a two-thirds probability that *at least one* module has an error.

offs towards getting the specification right. In many cases, a proof of "correct behaviour" provides information that could support proof of adequate speed or capacity. So the keys appear to be (1) correctness of the specification, and (2) correctness of the code with respect to the specification.

Proof could also eliminate much of the need for testing, and its considerable cost.[8] And tests don't ever tell us that code is correct — they can only tell us that either there *are* errors or that there does *not appear* to be errors. And *appearance* isn't enough, just because we didn't think of some test that would demonstrate an unimaginable kind of bug that just happens to be present.[9] So we would replace one cost by another — and we don't yet know which would be the greater — but would start to have the 100% confidence in our code which we never should have with testing.

2.2. Formal and Rigorous Methods in Mathematics and Engineering

So we need a mathematical method: and that means mathematical notation. A common misunderstanding is that mathematical notation exists as some sort of an end in itself, but this is actually the exact opposite of its purpose — as the eminent mathematician Hermann Weyl wrote in 1923:

> "Just as everybody must strive to learn language and writing before he can use them freely for the expression of his thoughts, here too there is only one way to escape the weight of formulae. It is to acquire such power over the tool ... that, unhampered by formal technique, one can turn to the real problem."[10]

Some of the proposed notations look superficially like traditional mathematics notations, some look more like program text: it doesn't matter which of those we choose so long as it frees us to express the necessary ideas in a way we can use. The current notations achieve that goal to a fair extent, in that they are usable but cumbersome: we need to do better.

Most Engineering disciplines have found classical Newtonian mathematics fitted their needs well for describing mechanical movement, fluctuations in fluid and

[8] Non-critical software projects typically report 30% to 40% of the total effort and cost going into testing: critical system testing costs even more.

[9] In the early 1970's I added a magtape driver to a disk-based operating system. It worked well until June 1975, when it seemed that interrupts seemed to "get lost". All through that summer it misbehaved, and we investigated hardware and software in vain for the fault: then the problem went away. It came back in June 1976: at which point we realized that this was a second hot summer: and that was our clue. It turned out that even in an air-conditioned computer room the tape drives got hot enough that the oil on them got thinner, the drives ran faster, and that there always had been a race between the hardware giving the interrupt and the software being ready to handle it, and now the hardware won the race just the couple of times in a thousand that made the tape drives unusable. In an air-conditioned room, who thinks to test program at different temperatures? Then there was the error I spotted over someone's shoulder in code that had worked well for several years: but that just happened to be the January of a leap year...

[10] This quotation could also be a motto for our efforts to design better programming languages, IPSE tools, GUIs...

electrical flows, etc. However, the Newtonian notion of "time" as being continuous doesn't correspond to the discrete notion of time in an imperative program, where "before" an instruction is a quite separate "time" than "after" the instruction. The non-classical nature of imperative "time" has been one of the motivations for the development of the functional languages, and the so-called "fifth generation" logic languages, both of which try to avoid any notion of time. [11] However, the functional and logic languages have new severe problems of their own[12], so it seems worth while to try to develop "discrete-timed" mathematics for imperative programming.

Instead of Newtownian mechanics, the part of traditional mathematics which seems most appropriate to our needs is "predicate logic", in which we make statements which we hope we will be able to "prove" as the results of theorems. Here's a couple of typical "predicate logic" statements about the arithmetic of natural (i.e. non-negative) integers:

– "There exists (at least) one integer which is the sum of all its devisors";
– "Every even number can be written as the sum of three squares".[13]

Writing those in a notation resembling programming language:

```
–   exists N in NATURAL =>
                (N = (sigma (I for (I in 1..N) and (N div I = 0))))
–   for all N in NATURAL, (exists A,B,C in NATURAL => 2*N=A
    ²+B²+C²) ¹⁴
```

These predicate logic statements about arithmetic are rather like the kind of "facts" (or in Mathematician's jargon, "assertions") which reflect our intent at points before and after instructions in an imperative program: e.g.

– "At this place in the program, elements 1 to N of the array A are in ascending sorted order"

which could be written in program-like notation as:

[11] But, for example, the "cut" operator in PROLOG introduces a sort of "try this, and *then* maybe try that" kind of time.

[12] For example, real-time PROLOG has turned out expensive, rather than successful.

[13] The first of those is certainly true: 6=1+2+3; 28=1+2+4+7+14; even 1=1! The second *appears* to be true if we include zero as a non-negative integer: $2 = 1^2 + 1^2$; $4 = 2^2$; $6 = 2^2 + 1^2 + 1^2$; $8 = 2^2 + 2^2$; $10 = 3^2 + 1^2$; $12 = 2^2 + 2^2 + 2^2$; $14 = 3^2 + 2^2 + 1^2$; $16 = 4^2$; $18 = 4^2 + 1^2 + 1^2$; $20 = 4^2 + 2^2$;... but no-one has either ever found a counter-example (despite searches up into the billions) or proven the statement "true".

[14] Notice the trick there to avoid defining "even-ness" — the statement has been notated more as "for every integer, twice the integer can be written...". I find the resulting notation more readable than

```
= for all N in NATURAL, ((exists M in NATURAL => N = 2 * M)
    => (exists A,B,C in NATURAL => N=A²+B² +C²)
```

which more directly translates the "for every even number" by expanding it into "for every number, if there is a number which it doubles, then...". From this we learn two things:

= a little deviousness is often needed to map English onto formal notations (perhaps because the English is vague? — see later...); and
= there are often multiple ways of notating an English version of a statement: it may not be easy to show they are equivalent; and one may turn out to be much easier to work with than another.

But programmers should be quite familiar with these phenomena, because they also arise in the transition from English specification/design into program code.

– `for all I in 1..(N-1), ( A(I) < A(I+1) )`[15]
If we associate a set of such "facts" with each relevant place in a program, then each instruction should correspond to the change between the set of facts we have "before" it, and the set of facts "after" it.[16]

So for the program as a whole we need a set of facts reflecting what should be true before executing it — i.e. a description of its environment, and what would be acceptable inputs — and a set of facts reflecting what should be true after executing it — i.e. a description of how its output relates to its input.

The same could apply to each routine and instruction of the program: the interrelationships of the sets of facts and the instructions themselves could then be used to "prove" that the program code is a correct implementation relative to the before-and-after fact-sets of the whole program.

Some facts might more naturally be associated with *data* at all times, rather than particular places in the *code* (i.e. specific times). The fact may concern a single variable (e.g. "this integer variable should always contain an even number in a particular range") or a combination of variables (e.g. "the sum of these two integer variables should always be even", or "the value of this integer variable always tells us what number of elements at the start of that array are currently in use"). The "always" in such a fact about variables usually will mean "from the time the variables are initialized until they cease to exist", rather than "from the time they start to exist...". These data-related facts aren't essential — but avoiding one will usually just mean replicating it at many instruction-related places.

Such sets of facts could be used while we do the designing to ensure that the design is correct; or to check a finished program; or some hybrid of those.

Whichever way we use the facts, there are a large number of proofs to make: and so we face the issue of how to make those proofs. Clearly, theorem-proving software to support this process is highly desirable: but certain features of what is needed (see later) place different requirements on this theorem-prover than those of the AI community. Currently, theorem-provers in general aren't in general as powerful as intuition-based proving of a human with a mathematical aptitude[17], so software to deduce *all* the sets of facts for a program (including between instructions) is not yet available. However, there are now a number of systems that can prove large classes of programs at source-code level by *completing* the sets of facts from human-provided key facts as "stepping stones".[18]

[15] For a more extensive example, see the specification part of the Appendix.

[16] In practice the full sets of facts can be very large, with many of them reflecting the effects of instructions well back, and not needed or made different for quite a distance. So if we were to write the sets down, we would have an unmanageably large text unless we carry forward these long-distance facts implicitly, and start writing them down explicitly again only just before instructions whose logic depends on the facts, or changes them.

[17] However, there are a few instances of spectacular proofs found by theorem-proving software, and which humans hadn't "spotted".

[18] From this point of view, the Mathematician's word "assertion" is more appropriate than my word "fact", because what we provide is a claim which we hope can be proven in that context.

Of course, the human-provided facts tend to be the few "hard" ones: what the software does is (1) remove the drudgery of the many easy ones, and (2) defend against the fallacies of logic to which humans are prone.

We can also draw another useful parallel from how traditional Mathematicians and Engineers work. As well as the full "formal method" presumed above, Mathematicians often skip easy parts of proofs with just a sketch of how they believe it could be completed. Likewise, Engineers often find that precise formulae reflecting what they are working with are intractable for design purposes, so instead they use "rule of thumb" formulae which are more conservative than the "real" formulae: as well as simplicity, the Engineer gains a valuable margin of error. These sketch-of-formal methods are called "rigorous".

In software construction, our opportunity to drop to "rigorous" methods rather than fully-formal methods arises from most critical software systems having only a few modules which are critical, and therefore needing fully-formal proof. Furthermore, we can often design so that a small number of simple modules need proof, instead of one complex module.[19] Clearly, the "needs proof" criterion has a subjective element, but clear guidelines (see M.o.D 1991b, for example) can be laid down to eliminate much of that subjectivity: such practices are quite normal in other Engineering disciplines.[20]

3. From User Need to Specification

Currently, specifications are usually written in English: but to be able to "prove" anything relative to a specification, we will need the specification in formal notation. Fortunately, that only applies to those aspects of the specification which relate to the parts of the program meriting "proof" (irrespective of whether the proof be formal or merely rigorous).[21]

[19] Further: if we can tolerate a warning when there is a malfunction, rather than assurance that there will never be a malfunction, we can often design so that a few simple modules validate the output of critical complex modules. For a naive example, checking a square root is much simpler than calculating it. Less trivial examples of "checking" being easier than "doing" include solution of partial differential equations and (as we have just noted) finishing a program-proof from human-provided key "facts".

[20] Both the difficulty of complete formal proofs of programs and failure to appreciate that sensible Mathematicians fall back from fully-formal methods only when appropriate have been used as perverse arguments that we should continue to use unreliable informal methods alone in program development: for example Demillo et al. 79 makes both arguments at once! Mankind would make little progress if we gave up on everything that's difficult.

[21] We have a risk here: from the specification we might proceed to a program design which calls for more of the specification to be formal... an expensive way of discovering which parts of the specification are critical.

3.1. Validation[22] of the Specification

As noted earlier, many projects run into trouble because the specification of the program defines a product which does *not* adequately satisfy the user's need, or loopholes which permit interpretations conflicting with the user's need. It could be that the analyst who studied the user requirement (or the user he consulted) failed to properly understand the requirement, or failed to express it; maybe the specifier misunderstood the requirements definition, conceived an inappropriate product, or mis-stated his specification of the product. So the specification can be vague, self-contradictory, incomplete, or just plain wrong. Common variations of "wrong" include that the requirements analyst or specifier implicitly depended on an invalid assumption, didn't know some relevant fact[23], or overlooked something he *did* know.

The most common tactic for validating a specification document is to give it to the user, and ask him if it he thinks it describes something would meet his need. The user is unlikely to be able to *fully* understand a specification in computer-jargon English, making the tactic dubious: but the user is very unlikely to understand anything from a formal specification[24], making the tactic totally inappropriate. So it seems paradoxical that a formal specification can facilitate validation in various ways:

– Vagueness is eliminated: apart from this being desirable in itself, removing "fuzziness" of English makes clashes and gaps more evident, so they are less likely to be written, and if they do get written they are more likely to be discovered (either by a human checker, or — not possible for English specifications — by software doing the checking).
– There is the possibility of generating a prototype from a formal specification, either by a program which inputs the formal specification and outputs source code of a prototype, or by a program that inputs the formal speci-

[22] The words "verification" and "validation" are used in this paper in the sense that "verification" means guaranteeing something in an absolute sense, but "validation" means guaranteeing something within real-world limits (e.g., against the ill-defined, as in this case).

[23] I once carefully specified a program with output on a colour VDU to be tolerant of red-green colour-blind users, but I'd never heard of orange-blue colour-blindness. I found out uncomfortably late.

[24] But Dandanell et al. 91 reports formal specification applications in the automation of the Meteor line of the Paris Metro, monitoring of ship engines, planning and design of communications nets, user interfaces for workstations, business transaction systems, monitoring of mining operations, and satellite-based geophysics experiments: only two of those (the networks and the UI) have users who are primarily computer oriented, but all appear to have users who have developed the necessary familiarity with formal methods. Similarly for O'Neill et al 88, where the application was control of jet engines for civil airliners. Mukherjee et al. 91 even reports a formal specification for a safety-critical situation with only a simulation intended *but no final computer application*. Where the user need is great enough, users *are* starting to invest in this capability, themselves. Another demonstration that formal methods are increasingly considered is that in one Proceedings of another conference *not* concerned primarily with formal methods we discover the first three papers — Peters et al. 90, Hird 90, Lees 90 — concern aspects of formal methods: was that a covert message from the program committee?

> fication and then itself simulates the specified product. A variation is more akin to expert-system methods: a program that inputs the formal specification and then answers questions about how the specified product will behave in response to given input: this last has the advantage that it will expose when the specification permits of alternative behaviours, whereas an "executable prototype" will only exhibit one of those behaviours — and which might not be the one which will take place for the final product.

And in any case, the formal specification can be back-translated into English, and that can be given to the user to approve.[25]

3.2. Problems even with formal specifications

With prototypes and back-translations, one can have a high confidence that the specification doesn't say anything contrary to what the user wanted, but it's still hard to be sure that a formal specification says enough: omission is still possible. It's also possible that the real world is too complex to model accurately in the specification: but that doesn't matter if the specification's mismatch with the real world is consistently conservative. Equally unfortunate, there's the insidious possibility that what the user wanted was misguided, and even dangerous in certain circumstances: in such a context a "safe" program would, in a threatening environment, *refuse* to behave according to its specification, because proven "correctness to the specification" is then exactly what one *doesn't* want.[26]

One problem with certain formal specifications, or parts of them, is that they may in effect be equivalent to the program. Obvious examples of this include specifications which require some Physical or Financial formula to be evaluated, or a Tax table to be looked up. In these cases — or at least in those parts of their applications — the need for formal specification seems slight.

A final and rather silly problem is that for some applications the formal specification may be much bigger than the eventual program! On the other hand, there are plenty of intuition-designed systems where the bulk of design documents far exceeds the bulk of the program listings. A bulky design in English has the pragmatic problem of its vagueness; a bulky formal specification (or design — see section 4) instead has the problem of unfamiliarity of notation; both have the difficulty in finding one's way around the document.

In short, formal specification avoids or lessens some problems with English specifications, but doesn't eliminate them.

[25] Advantages: the back-translated English version will be more precise (hence less to have gaps or clashes) than a version written first in English; disadvantages: the back-translated English will be more than ordinarily turgid, pedantic, and hard to read.

[26] Race 90 describes a case in which an unavoidable real-world mismatch turned out to be disastrously non-conservative in hard-to-foresee circumstances that) arose unfortunately early. See Holzapfel et al 88 for more discussion of this "safe-versus-correct" issue.

3.3. Notation of Formal Specifications and Assertions

There are two main styles of notation for formal specifications and assertions: maths-like and program-like.[27] The main maths-like notation is VDM (Jones 86): major descendants of it are Z (McMorran et al., 89) and the RAISE specification language, RSL (Eriksen et al., 91; George, 91). The main program-like notation is ANNA (Krieg-Brueckner 80, Luckham et al, 87): a major descendant of it is the SPARK Ada subset (Carre, 90).

The main criterion for choice between them is whether one is first and foremost a mathematician or a programmer: for most people that determines with which style they feel comfortable.[28]

Secondary criteria include

- the cost and availability of software tools to operate on the notation, and the hardware to display and print it: but the cost difference is small compared to the cost of the rest of the software one would want for serious formal manipulations;[29]
- how to relate a fragment of specification or other assertion to the resulting particular fragment of program text: embedding program-like assertions in actual program text seems natural, and usually can be achieved quite simply, but for maths-like assertions one tends to want separate documents for assertions and program, and hypertext-like pointers between the documents.[30]

A more fundamental difference between the two styles is that maths-like notation is almost always used to "build the product right" by supporting and checking each top-down refinement step in designing and coding the program, whereas program-like notations also are used for after-the-fact verification of previously-written code. Since real projects almost always have at least elements of bottom-up design (in the use of i/o libraries, if nothing else), and since there already exists a vast amount of code which merits tighter checking, the dual-purpose orientation of program-like notations seems more sensible to some people: others try to get the advantages of both styles by using maths-like notation to write a program whose

[27] There have been many attempts to impose other classifications on the notations and the systems underlying them, notably a division between systems which specify by saying "here is a model which the product should behave like" versus those which specify by saying "here is a test which we can use to recognise the product": but in practice the distinctions were blurred by the proposal of new systems which didn't fit nicely into the preconceived categories.

[28] Because the two forms of notation are equivalent in power, this is safer than the corresponding criterion for choice of a programming language for a given application that "I know one programming language, and will use it for this application with no regard as to whether the language is appropriate for the application".

[29] On the other hand, the book-production process of this paper makes maths-like notation problematic: accordingly, all the examples in this paper are ANNA-like.

[30] In ANNA and SPARK the embedding is achieved as comments which start with special "flag" characters. The program can then be submitted to a normal compiler, and the embedded annotations are ignored, whereas an ANNA- or SPARK-processor will ignore only "other" comments. The only restriction this embedding imposes on the programming language is that "normal" comments can't start with the flag characters — a minor restriction, and easy to live with.

code includes program-like annotations.

4. From Specification to Source Code

4.1. Ways of working

Those who use maths-like notation generally work by doing a top-down design of the program, always notating the designs in the formal notation, and at each step proving that the design defines a behaviour consistent with but more specific than that at the prior design step. Each individual step is called a "refinement" of the design. At some stage they need to move into program code: some do that as a last design/implementation stage with means to prove the program code reflects the final maths-notation design (or refines it); some make the move to program-like code (or pseudo-code) earlier, and then continue refining the maths-notation and code in parallel.[31]

At the earlier top-down stages, the maths-like notations seem to have more expressive power (i.e. more concise, more understandable once one is sufficiently familiar, and needing less deviousness to say what one wants to say). However, they have the problem that one will shift into a programming language sometime, so there's more to know and hence apparently higher training delays and costs than for a program-like notation: it's also been suggested that they demand higher-calibre staff.

Using program-like notation, there are two ways of working:

– starting with specifications in the annotated program-like notation, repeatedly refine program (both code/algorithm and data) and annotations in parallel until the program is executable (i.e. essentially similar to the method with maths-like notation, but without the problem of deciding of when and how to shift into program code) — this is called "formal design"; or
– write the program intuitively, in the orthodox way, then later add assertions and develop the proof of the program in terms of the assertions — this is sometimes called "program proof".

In short, that's choice between "build right" and "build, then later show right". These ways of working have different disadvantages:

– "build right" implies carrying a heavy additional burden throughout the design process;
– "build then show right" has the risk that when there is difficulty in finding a proof[32], it will often not be clear whether the problem is that the human or

[31] There's a hope that one day we will be able to write software that will translate a formal design into efficient program code. To some extent that's already been realized as "compilers" for "very high level languages" such as SETL — but the efficiency of the produced code is poor: the RAISE system apparently includes tools to generate Ada or C++ from finished designs in RSL, their formal notation. There's been more success in restricted application areas, e.g. generating certain parts of compilers.

[32] This form of proof is complicated by its being rather different from traditional mathematical proofs. In a traditional mathematical proof, one has a number of axioms which are generally accepted

software just hasn't found a proof, or whether the assertions doen't express enough information about a correct program, or whether the program is just plain wrong.[33].

In a sense, these two ways of working correspond respectively to developing the proof top-down versus developing it bottom-up.

For a previously-written program, a later attempt to fit a top-down proof to it is bound to be difficult, which tends inline us against the maths-like notations for such work, though the designers of RAISE have attempted to limit this problem for their RSL specification language.

Going in the other direction, certain aspects of the SPARK Ada-subset of the SPADE project were decided upon to make it a more suitable target language for designs in VDM and Z (if one wants to use SPARK only as the implementation language, rather than as the specification/design language as well).

4.2. The Target Language[34]

In what programming language will the program ultimately be implemented? Formal design all the way to assembler-level has been done, but subject to a new class of problems (see section 5) which then tangle into those discussed above. So the

as true (e.g. that if "a=b" then "b=a" — an alternative view is that this is part of the *definition* of "="), and a number of already-proven assertions: from the axioms and proven assertions certain deduction rules can be used to deduce further assertions which are considered "proven". In our case, we have a fixed set of axioms, but a set of assertions is local to a particular place in the program: assuming that assertion is true, we want to either prove the assertions at the next "place" true, or even deduce what the next set of assertions should be. The assertions at "the next place" don't need to be complete (usually, the "complete" set of assertions would be infinite) but they do need to be adequate that we can prove everything we need "at the next place". So at a sequence of places we want a sequence of sets of assertions (each really defining a probably-infinite set theorems), never losing information which will be needed later. When there is control flow, things get more complicated: for example at the end of an "if" statement, and at the end of its branches:

```
*if ... then*
...
$\{A_1, A_2, ...A_p\}$ – assertions at the end of the 'then' branch
*else*
...
$\{B_1, B_2, ...B_q\}$ – assertions at the end of the 'else' branch
*end if;*
$\{C_1, C_2, ...C_r\}$ – assertions after the 'if' statement
```

Here, we want to find assertions C_i which are implied by the end-of-then assertions *and also* by the end-of-else assertions, and which preserve all the information we will need later. In the extreme case that we want to lose *no* information, we must find the C_i so that collectively they lead to all the theorems which are in the intersection of the set of theorems deducible from the end-of-then assertions and of the set of theorems deducible from the end-of-else assertions: this is both most unlike orthodox theorem-proving *and* seems to lead us into a maze of undecidabilities and impossibilities related to Godel's theorem and Turing's hypothesis.

[33] Of course, a touch of orthodox testing may easily show the problem is that the program is wrong: but if the tests don't show symptoms of errors, then the problem remains.

[34] Carre 89a gives a longer (and rather different) discussion of this topic.

most common current approach seems to be to use some high-level language as a stepping-stone, keeping the two sets of problems apart.

That raises the issue of whether any current orthodox high-level language is suitable (perhaps in a suitably-annotated form): the short answer seems to be "not entirely". Criteria for a suitable language would include:

- simplicity;
- modularity (so we can build the proofs as combinations of many small theorems, i.e. avoiding "big" theorems for which we have exponentially-greater difficulty devising proofs);
- well-definedness (and machine-independence, in particular).

Some languages are clearly unsuitable: for example the language C was designed for low-level efficiency with convenience of use, and consequently leaves much as "what the CPU provides".

Pascal has some features that seem deceptively simple when described in English, but which turn out very imprecise when formally defined — "variant records" are the most notorious example (see Habermann 73 for more). It also leaves its character type and (more damagingly) its numeric types as "what the CPU provides". Original Pascal lacked large-scale modularization facilities: the recent "Extended Pascal" standard provides suitable facilities at the price of the simplicity of original Pascal.

Similar comments apply to Fortran — EQUIVALENCE is a major impediment to precise definition, numeric types are "what the machine provides", Fortran-90 provides large-scale modules at the cost of simplicity.

Ada is too big! — and so is not simple: consequently it's too involuted to define precisely (there have been expensive efforts which have come close, though). On the other hand, its numerics are unusually well-defined, and there are few machine-dependant features (but a number of implementor choices which the implementor must document). The modularization was designed in from the start, rather than added after the fact, and so is basically simple and well-integrated with the rest of the language (however, "extras" to the modularization facilities make the modularization seem complex: "extensive" is a more perceptive description). Ada was specifically designed for applications calling for a high degree of reliability, and so tends to what we need — but just too much of what we need. An early version of Ada (Ichbiah 79) even included an "assert" statement which was dropped from the eventual 1983 standard under pressure from "practical" programmers, but nonetheless still left behind a mentality close to what we need.

Eiffel (Meyer 92) may prove nearer to what we want:

- it clearly profits from the strengths and weaknesses of Ada (especially the strength of Ada's "design by contract" philosophy);
- it even (re)introduces assertions as part of the language (with a presumption that EIFFEL compilers will check whether programs conform to the assertions — but perhaps by runtime monitoring);

– but it introduces new language facilities (notably Object-Orientation) which cause additional difficulties for proof development (as if it wasn't already difficult enough!)

It also has the disadvantage of not yet being widely available, and in particular isn't yet much known in the "secure" programming community.

All of which leads us to the question of whether we can design a language with suitability for verification as a major criterion. A number of such languages have been attempted — e.g. EUCLID (Lampson et al. 77, London et al. 78), Gypsy (Ambler at al. 77) and NewSpeak (Currie 86) — but at the pragmatic programming level they all seem to be quirky, and so have not been widely used.

It seems then that we need a compromise: a common one is a subset of Ada. The SPARK Ada subset of the SPADE project (Carre et al 90) imposes quite severe restrictions on Ada: many eliminate implementor choices (such as of orders of execution in certain constructs) which the designers of Ada provided to enable the implementor to optimize, but other restrictions are clearly to reduce the complexity (and hence size and cost) of the tools needed to process SPARK programs. ANNA (Krieg-Brueckner et al. 80, Luckham et al 87) makes a brave attempt to provide ways of annotating even the highly machine-dependent constructs provided in Ada for bit-level control of peripherals, with the result that their toolset is not yet very extensive.[35]

5. From Source Code to Machine Code

Presuming that we do construct a verified program in an orthodox high-level programming language (whether by formal design or by informal-design-then-proof), we then need to obtain a correct machine code translation of the high-level source code. The ideal would be to have a formally verified compiler: correct input to correct translator means correct output (otherwise, the translator isn't correct).

Unfortunately, compilers are big and complex programs, so any proof or formal design of one will be correspondingly big. Nor is there an obvious way to make the "a few simple proven modules to check the rest" trick apply — most of a compiler is critical to the correctness of the translation it does.

Worse, to get an executable version of the compiler, we would need to translate it using another proven compiler (one for the language in which the compiler is written). So to get a proven compiler, we need another previously-proven compiler, for which we need... How do we get started?[36]

Finally, as we shall see in section 5.1, it is difficult to give precise definitions to instruction sets of real CPUs: how can we have a correct translation to something

[35] In the case of RAISE being used to generate Ada or C++, it is not clear how generous a subset of Ada or C++ is used.

[36] If the compiler is written in its own language (quite common for languages such as Pascal, Modula, Ada, C, EIFFEL, by means of a tactic called "bootstrapping"), we would need to compile it through its already-proven self in order to prove it! — an impossibility, unless we devise some non-obvious extension of the bootstrapping method.

ill-defined?

On a more optimistic note, there is one verified compiler from a simplified Pascal-like language to a realistic but fictitious machine-code — see Polak 81. So the "proven compiler" possibility seems worth trying for, despite the impediments.

If we accept that a proven translator is not currently viable, what can we do? The answer will be conditioned by the difficulties imposed by the machine code, so they are dealt with in the following sub-sections, before we turn to proposed solutions.

5.1. "DIFFICULT" FEATURES OF CPU INSTRUCTION SETS

At a superficial level, we can define the effect of an instruction in terms of "machine description language" (e.g., see Barbacci et al 77, 78, Joyner et al. 77). For example, Motorola 68000 instructions of the assembler form:

```
ADD.W   #n,Dr
```

to add a constant value n to a 16-bit ("Word") value in a data-register Dr could be defined by:

```
n + Dr -> Dr
```

but that's only a sketch:

– the 68000's D-registers are 32-bit, and the ".W" means that the least significant 16 bits of Dr will be used: what affect will the instruction have on the top bits of Dr? For example, if we are dealing with unsigned integers, will there be carry from the low half of Dr into the high half? or if we are working with signed integers, is n sign-extended?[37] If it is sign-extended, how do we say that in a usable formal notation?

– the 68000 has a number of condition-code registers which will be set, cleared, or left unchanged depending on the instruction and its result. For the ADD instruction, all are "changed according to the result of the addition": the Z condition-code register is set if the result is zero, the N condition-code register if the result is negative, the C condition-code register if there should be carry-out (into what? — see the last bullet item and the next paragraph), the V condition-code register if there is overflow (from what? the low 16 bits, or the full 32?) These effects can all be formally described in terms of the result, but the resulting description is large, complex, and generally unwieldy to work with. Motorola's manual tries to give a full semi-formalized description of the effects, but if one tries to write them in fully-formal notation a few subtle gaps show up.

One could describe condition codes in general as working by "side-effect": procedures with side-effects in high-level-language programs cause trouble both for informal and formal program development, so it's not surprising that side-effects

[37] Or would the effect of adding a negative n be achieved only by the SUB.W instruction?

of machine code instructions also cause trouble. In particular, if the program depends on "long" integer arithmetic, "carry" condition-codes cause special problems because their values get fed back into the data, whereas other condition-codes are usually used by "test and conditional branch" instructions, which don't (directly) mix the value of the condition-code with other multi-bit data, and so are "simpler" in a local sense.

Memory reference can also be hard to describe. For example, on the 68000, the memory reference in the instruction

```
    ADD.W    -(An),Dr
```

adds into Dr the value in that memory location whose address was in the An address register, *after having decremented the address in An.*[38] The indirection in this memory reference is rather fiddly to formalize, but the "autodecrement" side-effect is hard to describe, and involves particularly sensitive assertions to be generated and checked for the program under verification.

Where the options to use the bottom 16 or bottom 8 bits of a 32-bit 68000 register is problematic, the corresponding options on the Intel 8086 are worse. On the 8086, the AX register is 16 bits: the top half of AX can be accessed as "the AH register", and the bottom half as "the AL register". Likewise, the other two main registers, BX and CX, can be split into BH-and-BL, CH-and-CL. The aliasing in this is very difficult to formalize in a way that is easy to use. Compilers also have trouble with this, and often deal with it by crippling their register allocation — e.g. (1) not using any of AH, BH, CH, or (2) alternatively using AL and AH but not AX, using BX and CX but not BL, BH, CL, or CH. This is complicated by the 8086 having instructions which use AX, BX, and CX differently — for example the REP instructions make special uses of CX *and* of its halves.[39]

On almost all architectures various policies considered prudent when writing assembler by hand become even more important: e.g. strict separation of code and fixed data on the one hand from variable data on the other[40], and even strict separation of different classes of variable data become more important.

To a certain extent can reduce these problems by using only a subset of the CPU's facilities, the subset being chosen (where possible) to avoid "difficult" features. Unfortunately, nearly every CPU architecture has almost-unavoidable

[38] This form of address operand is useful for working through an array, element at a time; also for stack operation. One aspect that makes a formal description longer is that the increment or decrement is by the number of bytes in the operand (i.e. 2 or 4), rather than by a fixed value.

[39] And this is only part of the "clever" register architecture of the 8086. By contrast, the 68000 has a simpler and more generous register architecture, with 8 "D" registers, 8 "A" registers, and only one register having special properties (A7, which is presumed to be the stack register by the CALL and RETurn instructions). This exemplifies the wild variation between CPUs in "simplicity" versus "cleverness". (From this point of view of "niceness" or "regularity" of the instruction set, most RISC architectures are a mixture of "very nice" and "very nasty", when compared to orthodox architectures.)

[40] The "self-modifying code" which was normal practice on early machines — which manifested itself in COBOL as the "ALTER GOTO" verb — would cause terrible difficulties for verification. On the other hand, compiler writers often decide to generate code with very murky characteristics.

"difficult" features: for example, avoiding "autoincrement" and "autodecrement" memory references on the 68000 will only complicate the program in other ways (but other "clever" forms of 68000 memory reference *are* more easily avoidable). On almost all machines, interrupt-handling is "*very* difficult" to formalize (both its asynchronous nature, and the interrupt-priority regime), and usually impossible to avoid in real programs.

"Subsetting" the CPU also has the problem that we then either have to hand-write the desired program in assembly language, or have a special compiler which only generates machine code within the subset: both options are expensive, in different ways.

In short, "clever" CPU features designed to help a human writer of assembly-language programs become a nuisance for compiler-writers, and an extreme nuisance for formalization. For formalization, the ideal would be a target-machine instruction set that is simple, and which has a modest-sized formal definition.

5.2. "Difficult" features of assembly languages

Hand-written machine-code programs are usually notated in a textual form called "assembly language", rather than directly in binary. When there is need to verify machine code, the verification is often done at assembler level, because the binary cannot reliably be read by humans, and lacks essential information for human or software to use (see next subsection).

The "assembly-language" text is translated to binary by an "assembler" program. Just as we had the problem of "trustworthy" compilers, so we have the problem of trustworthy assemblers.

In cases where one line of assembly-language text is translated to one machine code instruction, the assembler is simple, and we have the possibility of verifying it (or rigorously designing it) fairly straightforwardly, barring the difficulties discussed in the last sub-section. So if assembly-level language is to be used as a stepping-stone in the development of a verified program we will tend to prefer a minimal-but-adequate assembly language and assembler.

Most assemblers support language features that are *not* simple, as aides to human writers of assembly-language programs: for example, such features often include macro-generation, conditional compilation, and means to provide multiple names ("aliases") for data locations — all traditionally heavily used by assembly-language programmers, and all complex. The corresponding assembler would be hard to verify, so if such a language were to be used for a program to be verified, it would probably be better to separate the assembler into two: a prepass into "simple" assembly-language, which would then be verified, then a simple assembler for translation into binary. Suitable software wouldn't be particularly expensive to write, but is seldom available.

One might hope that a back-translation strategy could verify the translation of a particular program: that is, the program could be translated from assembly language to binary, and then "disassembled" back into source which would be compared with the original assembly source, supposedly verifying the binary version on the argument that if the original source version was correct but the binary version was incorrect, then the back-translated source would be incorrect and therefore different from the original. This has obvious fallacy that if the assembler and disassembler programs were written with the same misunderstanding then the assembler would translate wrongly, and the back-translation from the disassembler would repair the error. Despite the fallacy, this is a square wheel that has been re-invented a number of times.[41]

5.3. The Lack of Information in Assembler and Machine Code

The compilation process discards a lot of information that was present or latent in the high-level language source: for example, given some inter-related branch instructions in the translated code, so they reflect some meaningful part of the source control flow, or were they due to an optimization of an "if" test of a complex condition?

This makes it more difficult to prove a low-level version of a program than the high-level source version, even without the difficulties of section 5.1. Additional information would therefore be helpful.

As already noted, within the two forms of low-level code, machine code has had even more information discarded than assembly-level code. But this time, we could hope to carry assertions and proof forward from a one-instruction-per-line assembly language to binary, and use them to audit the binary version.

5.4. Compile, with Proof

An obvious tactic is then to carry forward the formal design or proof of the high-level language version of a program. Instead of permitting information to be discarded by the compilation process, the compiler would preserve any desirable information from source level, perhaps as additional assertions: for example

– initialization of variables (made more difficult in assembler by loss of "scope" information);
– that values of a certain variable should be in a range associated with its type or subtype (made difficult by loss of information on the type or subtype);
– ensuring the run-time program won't try to "follow down" any "nil" pointer-values (made difficult by loss of "type" information — in this case of information about "pointer" types);

[41] There is an extra problem that the process of assembly discards information — e.g. the layout of the input, what name (if any) was associated with a particular address, etc.: so the comparison would need to be tolerant of the resulting discrepancy. The various reinventions have included a variety of dubious patches to supposedly overcome this problem.

- array indexes being in bounds (made more difficult in assembler by loss of explicit information on the bounds — so its really the bounds that would survive as additional assertions);
- which branch instructions reflect what kinds of source-level flow-of-control constructs;

and so on.

Compilation can even help, because many of the analyses done during compilation (especially for optimization — see Carre 89b) yield information which is useful or needed for the proof. 'Also, there is the possibility that if the proof couldn't be completed at source level because some assertion doesn't follow from the source (e.g. that a certain variable will be able to hold values in a certain range), the compiler could translate in such a way that the assertion will be true for the translated code (e.g. that the variable be chosen to be of a size adequate to hold the values of the desired range).

Unfortunately, even modest optimizations will re-order code, merge code, change the organization of the data, and so on: and all of those will have the effect that source assertions will have no-where to go, and new assertions will be needed: as a result the carried-forward proof will at least need partial re-proving. The alternative is to go to painful lengths to not optimize at all: which is exactly what many safety-critical projects have done.

Even then, most compilers need to generate low-level code which take advantage of CPU-specific "tricks", making the output unavoidably hard to verify.

Unfortunately, this "compile, carrying forward the proof" approach has been talked about more than done. The main reason for this seems to be that it requires much-modified compilers, and that no-one has seen fit to fund such an expensive and risky venture.

5.5. Directly prove low-level code

This has been done, both for formally-developed code refined all the way to assembler level and also for hand-written code.

The SPADE project provides two well-documented examples. The essential method of each was that machine-code subsets (avoiding the "hardest" features of the CPUs) are translated by a program written in Prolog to FDL (the modelling language of the SPADE proof checker) together with manually-supplied assertions. Clutterbuck et al., 88, used this method for a subset of the 8080 processor: O'Neill et al., 88, used it for a modularized assembler language for the A8002 processor[42]. A new assembler-to-FDL translator is needed for each new assembly language, but thereafter fairly standard tools are used.

One might worry about the correctness of the translators to FDL, being manually written and hard to verify: somewhat less about the more-used SPADE proof

[42] The A8002 application was control of the Rolls-Royce R-111 "524 series" engines now in use in Boeing 747 and 767 and other airliners: the 8080 project was not for a specific application.

checker. However, if either of these had an error, it is extremely probably that the result would be that a proof wouldn't be found for a correct program: that is, such errors are almost certainly fail-safe. Although such errors were apparently a problem, the greatest problem was the substantial amount of effort that was needed to develop the very large number of highly-detailed assertions that were needed: this because low-level versions of programs are longer and more detailed than the corresponding high-level versions.

5.6. Translate to Intermediate Level, Prove, Proven Translation

>From all the above, it seems that there could be hybrid solutions to the problem of converting verified high-level programs into verified low-level equivalents. In particular, there is the possibility that can identify some intermediate level, such that it is still sufficiently high-level that the proof for formal design can be carried forward to it relatively easily, but sufficiently low-level that a verified translator from it is a realistic proposition.[43]

There appear to have been two proposals of this sort in recent years. Wichmann, 89, proposes a 'Low Ada' language — essentially a simplified subset of Ada, to which verified full Ada could be transformed. To a large extent the high-level features which survive in "low Ada" are (1) the control constructs (which we have already observed would need to be preserved in some form, to facilitate the low-level proving) and (2) much-simplified assignments (it being presumed that all the checking normally done in full Ada assignments would be done explicitly in Low Ada). So Low Ada attempts to stay as provable as possible, but is still rather far from machine code. Lester et al., 91, makes establishing a desirable level of the intermediate language something to be determined, but tends to presume something like the intermediate languages commonly used in multi-targeted compilers, i.e. a somewhat lower level than Low-Ada. Compiler-like intermediate languages have the advantage of there being a substantial literature on algorithmic methods of translating them to machine-level (Cattell 78, Cattell et al. 79, Cattell 80, Fraser 77, Glanville 77a, Glanville 77b, Graham et al. 78, Lester 82, Wasilew 72). If one has confidence in the algorithm of one of these methods, the correctness of a translation using the method depends only on the machine description given to the method. A weakness of these algorithmic methods is that they generally will choose a correct set of machine instructions, and most place them into a correct

[43] This may not be possible: "easily proven" might imply a higher level than we can verifiably translate from, given current verification methods. Although the possible problem in this case is a gap in the levels, it may be that we can gain intuition from a related problem of an overlap of levels: in the early days of compiling technology, there was an attempt to find a language low enough to be a common target for a variety of high-level languages and a common source for a variety of machine-codes (Conway 58, Strong 58, Steele 61) but even with the limited variety of high-level languages and machines at that time the "highest common denominator" of the languages proved lower than the "lowest common multiple" of the machines (Steele 63: 'UNCOL' stood for 'UNiversal Computer-Oriented Language). There are many lessons to be learned from the "UNCOL experience".

order, but all are very weak on register-allocation aspects of the translation to machine-code. For example, Leverett 91 discusses register-allocation in the same "Compiler-compiler" system as Cattell, with the two parts of that system apparently being very separate: Lester's method can treat separate registers as distinct machine code entities, and so can produce locally-best code at the cost of a search which takes time exponential in the number of registers. As it stands, the algorithmic methods can therefore either produce poor-but-correct machine code, or will need extension to deal with register allocation. It seems likely that different register architectures (e.g. the 68000's two banks of registers versus the 8086's no-two-alike registers) will involve quite different extensions to the underlying algorithms, thereby weakening confidence in the extended algorithm (because it starts to be complex, and a particular extension will be less-used and so less-scrutinized) and in the code implementing it.

6. Conclusion

The first time a programmer attempts a "rigorous" verification of any program code, he is likely astonished by two things: (1) the precision needed; (2) the sloppiness of the intuitive logic which had been used in writing the program (and which previously was probably thought impeccable). Intuitive program construction is just not adequate for important programs — we make too many implicit and unjustified assumptions (see also Harel 80).

If we "prove" or "formally develop" a program or program component, we gain a level of confidence in it that is unavailable with intuition backed with testing, but it is crucial to prove the software *all the way to the executable machine code version*. Software such as RAISE and SPARK/SPADE for facilitating the proofs is becoming available.

And then we'll systematically test the program anyway — but not nearly as expensively as in our traditional "extreme experimental" manner.[44]

We should not be surprised or disturbed by the need for a mixture of proof tactics — corresponding sorts of mixture (perhaps different mixtures from project to project) are common in other Engineering disciplines. The opportunity to Engineer programs is there. All we need is the common sense and courage to make the investment needed to drag ourselves out of the 1960's.

Acknowledgements

The topic of this paper is vast, having been the subject of much work (albeit scattered) over the last thirty years: consequently, it has not been possible to cover the whole issue in only a single tutorial paper. This required that reportage

[44] Consider the differences between how a Civil Engineer will test a bridge that was rigorously designed, versus a bridge that was "designed" only by intuition... Systematic test of an intuition-defined entity is a much larger and guess-based task than testing a rigorously-designed entity.

be arbitrarily selective and unduly abbreviated (especially in the references): so I would like to give a general acknowledgement to the many whose work I otherwise be acknowledging in detail.

However, I would like to make particular acknowledgements to Peisong Huang and Dr Brian Wichmann for valuable discussions on the topic of the correctness of executable programs and how to achieve it, and to Mike Woodger for having, over the years, opened my eyes to the inadequacies in how we often do things. I must also thank Mike for identifying the Turing quotation.

References

Ambler A.L., Good D.I., Browne J., Burger W.F., Cohen R.M., Hoch R.M. and Wells R.E., 1977: 'Gypsy: a language for specification and implementation of verifiable programs', in SIGPLAN notices 12:3 (Proceedings, ACM conference on language design for reliable software), pp 1-10.

Barbacci M.R., Barnes G., Cattell R.G.G., and Siewiorek D., 1977: 'ISPS reference manual', Carnegie-Mellon University report CMU-CS-TR-79-137.

Barbacci M.R. and Parker A., 1978: 'Using emulation to verify formal architecture descriptions', IEEE "Computer" magazine, May 1978, pp. 51-56.

Carre B.A., 1989a: 'Reliable programming in Standard Languages', chapter 5 of Sennet C. (ed.), *High Integrity Software,* Pitman, London.

Carre B.A., 1989b: 'Program analysis and verification', chapter 8 of Sennet C. (ed.), *High Integrity Software,* Pitman, London.

Carre B.A. and Garnsworthy J., 1990: 'SPARK — An Annotated Ada Subset for Safety-Critical Programming', in *Tri-Ada '90 Proceedings,* ACM Press, New York, pp. 392-402.

Cattell R.G.G., 1978: 'Formalization and automatic derivation of code generators', Carnegie-Mellon Univerity report CMU-CS-TR-78-115 and AD-A058-872/3WC.

Cattell R.G.G., Newcomer J.M., and Leverett B.W., 1979: 'Code-Generation in a machine-independent compiler', SIGPLAN notices (Proceedings, Symposium on Compiler Construction) 14:8 pp. 65-75.

Cattell R.G.G., 1980: Automatic Derivation of Code Generators from Machine Descriptions', ACM Transactions on Programming Languages 2:2, pp.173-190.

Clutterbuck D.L., and Carre B.A., 1988: 'The verification of low-level code', in Software Engineering Journal, May 1988.

Conway M.E., 1958: 'Proposal for an UNCOL', Communications of the ACM, 1:10 pp. 5-8.

Currie I.F., 1986: 'NewSpeak — an unexceptional language', Software Engineering Journal 1, pp 170-176.

Dandanell B., and George V., 1991: 'The LaCoS Project', Computer Resources International A/S, Birkerod, Denmark.

De Millo R.A., Lipton R.J. and Perlis A.J.: 1979, 'Social Processes and Proofs of Theorems and Programs', CACM 22(5), pp. 271-280. See also the resulting letters, CACM 22(11) pp. 621-630 and CACM 23(5) pp. 307-308, especially the refutations by Lamport, Maurer and Leverett.

Eriksen K.E., and Prehn S., 1991: 'RAISE Overview', Computer Resources International A/S, Birkerod, Denmark.

Floyd R.W., 1967: 'Assigning Meanings to Programs', in Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, ed. J.T. Schwartz, American Mathematical Society, Providence R.I. pp 19-32.

Fraser C.W., 1977: 'Automatic Generation of Code Generators', Yale University Department of Computer Science.

George C. (ed.), 1991: 'The RAISE Specification Language', Computer Resources International A/S, Birkerod, Denmark.

Glanville R.S., 1977a: 'A new method for Compiler Code Generation', Univerity of California at Berkeley Department of Computer Science.

Glanville R.S., 1977b: 'A Machine-Independent Algorithm for Code Generation and its Use in Retargetable Compilers', Univerity of California at Berkeley Department of Computer Science.

Graham, S., and Glanville R.S., 1978: 'A new method for Compiler Code Generation', Proceedings of the 5th ACM Symposium on the Principles of Programming Languages, pp. 231-240 (this is a review of Glanville 77a).

Habermann A.N., 1973: 'Critical comments on the programming language Pascal', Acta Informatica 3, pp. 47-57.

Harel D., 1980: 'On Folk Theorems', CACM 23(7), pp. 379-389; also Denning's follow-up, CACM 23(9), pp.493-494.

Hird G.R., 1990: 'Towards Reuse of Verified Ada Software', in *Tri-Ada '90 Proceedings,* ACM Press, New York, pp. 14-21.

Holzapfel R. and Winterstein G., 1988: 'Ada in Safety Critical Applications', in *Proceedings of the Ada-Europe Conference,* Munich, published by Cambridge University Press.

Ichbiah J.D. et al., 1979: 'Reference Manual for the Programming Language Ada', SIGPLAN Notices 14:6 (June 1979), Part A.

Joyner, Carter, and Brand, 1977: 'Using machine descriptions in program verification', IBM research report RC 6922.

Jones C.B., 1986: 'Systematic Software Development using VDM', Prentice-Hall.

Krieg-Brueckner B. and Luckham D.C., 1980: 'ANNA: towards a language for Annotating Ada Programs', SIGPLAN notices 15:11 (Nov 1980) pp. 128-138.

Lampson B.W., Horning J.J., London R.L., Mitchell J.G. and Popek G.L., 1977: 'Report on the programming language EUCLID', SIGPLAN notices 12:2.

Lees R.A., 1990: 'A Tailored Design Language: Putting Model Based Formal Specification into Practice', in *Tri-Ada '90 Proceedings,* ACM Press, New York, pp. 22-31.

Lester C., 1982: 'Some tools for automating the production of compilers and other translators', Imperial College Department of Computer Science, University of London.

Lester C. and Huang P., 1991: 'A fresh approach to program proving (from Ada to machine code)', Portsmouth Polytechnic School of Information report TR-91/3.

Leverett B.W., 1981: 'Register Allocation in Optimizing Compilers', Carnegie-Mellon University report CMU-CS-81-103.

London R.L., Guttag J.V., Horning J.J., Lampson B.W., Mitchell J.G. and Popek G.L., 1978: 'Proof rules for the programming language EUCLID', Acta Informatica 10:1, pp 1-26.

Luckham D.C., von Henke F.W., Krieg-Brueckner B., and Owe O., 1987: 'ANNA — a language for Annotating Ada Programs', Lecture Notes in Computer Science 260, Springer-Verlag.

McMorran M.A. and Nicholls J.E., 1989: 'Z User Manual', IBM United Kingdom Laboratories report TR12.274.

Meyer B., 1992: 'EIFFEL: the language', Prentice-Hall.

M.o.D., 1991a: Interim Defence Standard 00-55, 'The Procurement of Safety Critical Software in Defence Equipment', (British) Ministry of Defence, April 1991.

M.o.D., 1991b: Interim Defence Standard 00-56, 'Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment', (British) Ministry of Defence, April 1991.

Mukherjee P. and Stavridou V., 1991: 'The formal Specification of Safety Requirements for the Storage of Explosives', report DITC 185/91, National Physical Laboratory, Teddington, England.

O'Neill I.M., Clutterbuck P.L., Farrow P.F., Summers P.G. and Dolman W.C., 1988: 'The Formal Verification of Safety-Critical Assembly Code' in *Safety of Critical Control Systems (Proceedings of IFAC-SafeComp 88),* pp. 115-120.

Peters J. and Hankley W., 1990: 'Proving Specification of Tasking Systems Using Ada/TL', in *Tri-Ada '90 Proceedings,* ACM Press, New York, pp. 4-13.

Polak W., 1981: 'Compiler Specification and Verification', Lecture Notes in Computer Science 124, Springer-Verlag.

Race J., 1990: 'Computer-encouraged pilot error', in Computer Bulletin, Aug 1990, pp 13-15.

Steele T.B., 1961: 'A first version of UNCOL', Proceedings of the Western Joint Computer Conference 19: pp. 371-378.

Steele T.B., 1963: 'UNCOL — the myth and the fact', Annual Review of Automatic Programming

2, pp. 325-344.
Strong G., 1958: 'The problem of program communication with changing machines: a proposed solution', Communications of the A.C.M. 1:8 pp. 12-18 and 1:9 pp. 9-15.
Turing, A., 1949: 'Checking a large routine', talk to the Inaugural Conference of EDSAC, proceedings published by the Mathematical Laboratory, Cambridge University, 1950. There is also evidence that Turing and others considered these issues at Princeton University in the 1930's.
Wasilew, 1972: 'A compiler writing system with optimisation capabilities for complex order structures', North-Western University and DAI 72-32604.
Weyl H., 1923: "Raum, Zeit, Materie", Berlin.
Wichmann B.A., 1989: 'Low-Ada: an Ada Validation Tool', Report DITC-144/89, National Physical Laboratory, Teddington, England.

APPENDIX: SPECIFICATION AND PROOF OF TWO PROCEDURES

The appendix gives specifications and proofs for two simple procedures in Ada, using a simplified ANNA-like notation. "Simple" turns out to be not-so-simple...

First: A SPECIFICATION of pre- and post-conditions for what the Ada Standard rather completely calls a 'procedure specification':

```
procedure SWAP(A,B:in out ELEMENT);
--| out     A = in B
--|     and B = in A
```

Text after --| is assertion: Ada comments start with "--" and continue to the end of the line, so the assertions would be ignored if the above and following texts were input to an Ada processor (as opposed to an ANNA-like processor). In the following, Ada-style comments are used inside assertions to comment on the assertions. For a specification, assertions preceded by "in" are preconditions, i.e. should be true before any call of the procedure; and assertions proceeded by "out" are postconditions, i.e. will be true for a given procedure call if the preconditions are met for that call and if the body of the procedure is correct. This procedure has a null precondition — that is, it can be called anywhere with no precondition needing to be met. (There is, however, the implicit precondition that any data or variables going "in" to a procedure will have been initialized.)

Now for an Ada body for that Ada specification, with a "proof" of the correctness of the body:

```
procedure SWAP(A,B:in out ELEMENT) is
        --| {A = in A, B = in B}
    OLD_A : constant ELEMENT := A;
        --| OLD_A = in A
        --| {A = in A, B = in B, OLD_A = in A}
begin
        --| {A = in A, B = in B, OLD_A = in A}
    A := B;
        --| { A = prior B }
                -- deduce { A = in B } (because {prior B = in B}),
                -- so a summary of what we know here is now:
        --| {A = in A, B = in B, OLD_A = in A} - {prior A=in A}
        --|                                      U {A=in B}
```

```
                -- i.e.
        --| {B = in B, OLD_A = in A, A = in B}
    B := OLD_A;
        --| { B = prior OLD_A }
                -- deduce { B = in A }, summary "here" is:
        --| {B = in B, OLD_A = in A, A = in B} - {prior B=in B}
        --|                                      U {B=OLD_A,B=in A}
                -- i.e.
        --| {OLD_A = in A, A = in B, B = OLD_A, B = in A}
end SWAP; -- "forgets" any facts involving locals
    --| {OLD_A = in A, A = in B, B = OLD_A, B = in A} - {OLD_A}
            -- i.e.
    --| {A = in B, B = in A}  -- Q.E.D.
```

Clearly, the sets of facts we will need to carry round in the text can get very big, so if we carry on with a notation like that it will become unreadable: and that defeats the whole purpose of notation ("to provide means to express some class of concepts concisely").

Perhaps the need is really for program tools to lessen that problem. For example, if we only write "stepping stones" — the things to deduce from (on entry), to deduce to (on exit) and the "deductions" (along the way) then we might input

```
procedure SWAP(A,B:in out ELEMENT) is
        --| {A = in A, B = in B}
    OLD_A : constant ELEMENT := A;
        --| OLD_A = in A
begin
    A := B;
        --| { A = in B }
    B := OLD_A;
        --| { B = in A }
end SWAP;
    --| {A = in B, B = in A} -- Q.E.D.
```

to have the tool output the along-the-way summaries

```
procedure SWAP(A,B:in out ELEMENT) is
        --| {A = in A, B = in B}
    OLD_A : constant ELEMENT := A;
        --| {A = in A, B = in B, OLD_A = in A}
begin
        --| {A = in A, B = in B, OLD_A = in A}
    A := B;
        --| {B = in B, OLD_A = in A, A = in B}
    B := OLD_A;
        --| {OLD_A = in A, A = in B, B = OLD_A, B = in A}
end SWAP;
    --| {A = in B, B = in A} -- Q.E.D.
```

or even maybe just output "Q.E.D.".

Alternatively, we could number or letter the places that assertions are needed, and then present the assertions separately. Being able to identify a particular assertion also makes it easy to provide an audit trail of what another assertion was deduced from: the second example does that.

Many assertions would be very long if written in full. So we "factor out" fragments that would repeatedly appear in different assertions. This also has the advantage that we can name the factored-out fragments, so that both the statement of the fragment and of its uses become more understandable. In ANNA this is done as "virtual text", preceded by "--:". The virtual text is also written in Ada-like code, as if it were to be executed as part of execution of an assertion which should return `TRUE` at runtime.

Some virtual text, with an Ada declaration which the virtual text depends on:

```
type ARR is array (INDEX) of ELEMENT;
                -- the specification AND the virtual text will
                -- operate on this..

--: function ASCENDING (Y:ARR) return BOOLEAN is
--: begin
--:     return (for all I,J in Y'RANGE, I<J implies Y(I)<Y(J) );
--: end ASCENDING;

--: function UNIQUE (Y:ARR) return BOOLEAN is
--: begin
--:     return (for all I,J in Y'RANGE, Y(I)=Y(J) implies I=J);
--: end UNIQUE;

--: function IS_PERM(Y,Z:ARR) return BOOLEAN is  -- this should only
--: begin                                        -- be used where the
--:     return (for all I in Y'RANGE,            -- ranges of Y and Z
--:               exists J in Y'RANGE => Y(I)=Z(J) ); -- are the same.
--: end IS_PERM;
```

Ideally, there would be some way of telling the tools that the following "invariants" should all be `TRUE`, at all times:

```
Y.1:  IS_PERM(X,X) for any initialized array X or type ARR
Y.2:  UNIQUE(Y) and IS_PERM(Y,Z) implies UNIQUE(Z)
Y.3:  ASCENDING(X(I,I))   -- a one-element "slice" of X is "ascending"
Y.4:  ASCENDING(X(I,I-1)) -- a null slice is (vacuously) ascending
Y.5:  P(A,B) followed by SWAP(A,B) implies P(B,A) for any proposition P
```

Ideally the tools should check the correctness of these "invariants" from the virtual code and the postconditions of `SWAP`, and should complain if `IS_PERM` is misused (i.e. in a context where its parameters are not provably of the same length).

Now for the SPECIFICATION of pre- and post-conditions for the second Ada 'procedure specification':

```
procedure SORT (X:in out ARR);
--| in  UNIQUE(X)
--| out UNIQUE(X) and IS_PERM(in X,X) and ASCENDING(X)
```

The "`in X`" in the postcondition means "the (composite) value of X when the procedure was entered", so `IS_PERM(in X,X)` means that the values of the elements after `SORT` are a permutation of the elements before.

Now for the same Ada body for that Ada specification, with a "proof" of the correctness of the body:

```
procedure SORT (X:in out ARR) is
begin
    {A.i}
    for I in INDEX'FIRST + 1 .. INDEX'LAST loop
        {B.i}
        for J in reverse INDEX'FIRST+1 .. I loop
            --| {C.i}
            if X(J) < X (J-1)
            then
                --| {D.i}
                SWAP (X(J),X(J-1));
                --| {E.i}
            else
                --| {F.i}
                null;
                --| {F.i} -- (still)
            end if;
            --| {G.i}     -- It seems to be a matter of taste
        end loop;         -- whether this text becomes "more
        --| {H.i}         -- readable" if the assertions are
    end loop;             -- placed on the ends of the prior
    --| {I.i}             -- lines, making the text "short and
end SORT;                 -- fat" instead of "long and thin".
```

```
{A.i}:
   A.1:  UNIQUE(in G)               -- precondition
   A.2:  IS_PERM(in X,in X)         -- invariant Y.1
   A.3:  ASCENDING(X(X'FIRST..X'FIRST-1)) -- invariant Y.4
{B.i}:
   B.1:  I in X'FIRST+1..X'LAST     -- the "for" range
   B.2:  UNIQUE(X)                  -- A.1 and H.2
   B.3:  IS_PERM(X,in X)            -- A.2 and H.3
   B.4:  ASCENDING(X(X'FIRST..I-1)) -- A.3 and "for" iteration from H.4

{C.i}:
   C.1:  J in X'FIRST+1..I          -- the "for" range
   C.2:  I in X'FIRST+1..X'LAST     -- B.1
   C.3:  UNIQUE(X)                  -- B.2 and G.3
   C.4:  IS_PERM(X,in X)            -- B.3 and G.4
   C.5:  ASCENDING(X(X'FIRST..J-1)) -- B.4 and "for" iteration from G.5
   C.6:  ASCENDING(X(J+1..I))       -- Y.4 and "for" iteration from G.7
   C.7:  X(J-1) < X(J+1)             -- vacuous on first iteration, and
                                     -- "for" iteration from G.6
   C.8:  X(J) < X(J+1)               -- vacuous on first iteration, and
                                     -- "for" iteration from G.8

{D.i} = {C.i} U the if-condition
   D.1..8 = C.1..8
   D.9:  X(J) < X(J-1)               -- the if-condition

{E.i} = {D.i} then SWAP (X(J),X(J-1))
   E.1:  J in X'FIRST+1..I           -- = C.1, D.1
   E.2:  I in X'FIRST+1..X'LAST      -- = C.2, D.2
   E.3:  UNIQUE(X)                   -- from C.3=D.3 and SWAP
   E.4:  IS_PERM(X,in X)             -- from C.4=D.4 and SWAP
   E.5:  ASCENDING(X(X'FIRST..J-2))  -- from C.5=D.5 less damage by SWAP
   E.6:  X(J-2) < X(J) [=old X(J-1)] -- from C.5=D.5 and SWAP
   E.7:  ASCENDING(X(J+1..I))        -- from C.6=D.6
   E.8:  X(J) [=old X(J-1)] < X(J+1) -- from C.7=D.7 and SWAP
   E.9:  X(J-1) [=old X(J)] < X(J+1) -- from C.8=D.8 and SWAP
   E.10: X(J-1) [=old X(J)] < X(J) [=old X(J-1)]
                                     -- from D.9 and SWAP

{F.i} = {C.i} U the inverse of the if-condition
   F.1..8 = C.1..8
   F.9:  not { X(J) < X(J-1) }       -- inverse of the if-condition
   F.10: X(J) >= X(J-1)              -- more tractable form of F.9?
   F.11: X(J-1) < X(J)               -- from F.3 and F.10
   F.12: ASCENDING(X(X'FIRST..J))    -- from F.5 and F.11: also subsumes
                                     -- F.5 and F.11

{G.i} = {E.i} intersection {F.i}     -- see (*) below
   G.1..4 = A.1..4                   -- Identical in {E.i} and {F.i}
   G.5:  ASCENDING(X(X'FIRST..J-2))  -- =E.5, and implied by F.5=C.5
   G.6:  X(J-2) < X(J)               -- =E.6, also implied by F.12
   G.7:  ASCENDING(X(J..I))          -- implied by E.7 together with E.8,
                                     -- also by F.6 together with F.8
   G.8:  X(J-1) < X(J)               -- =E.10, also implied by F.12
```

(*) It's not the sets of assertions that the intersection applies to, but the sets of theorems they generate! — see the main paper. By contrast, the union of sets of assertions is a set of assertions defining the set of theorems which is the union of the two sets of assertions for the original sets:

$$\text{THEOREMS}(\{X_i\}) \cup \text{THEOREMS}(\{Y_i\}) = \text{THEOREMS}(\{X_i\} \cup \{Y_i\})$$

```
{H.i} = {G.i} the last time round the J-loop, i.e. with J replaced
                                                   by (X'FIRST+1)
   C.1 = G.1, carried forward, becomes irrelevant (since J no longer exists)
   H.1 = B.1 = G.2
   H.2 = B.2 = G.3
   H.3 = B.3 = G.4
```

```
H.4:  ASCENDING(X(X'FIRST+1..I))     -- replacement in G.7
H.5:  X(X'FIRST+1-1) [i.e. X(X'FIRST) < X(X'FIRST+1)
                                     -- replacement in G.8
H.6:  ASCENDING(X(X'FIRST..I))       -- implied by H.4 and H.5, which
                                     -- it subsumes
```

After the replacement, G.5 becomes `ASCENDING(X(X'FIRST..X'FIRST-1))`, and G.6 becomes `X(X'FIRST+1-2) < X(X'FIRST+1)`, both of which are vacuous, and so can be struck out at this point in the program (and introduced anywhere it is needed!)

```
{I.i} = {H.i} the last time around, i.e. with I replaced by X'LAST
   H.1 becomes irrelevant
   I.1:  UNIQUE(in G)                 -- H.2 = A.1
   I.2:  IS_PERM(X,in X)              -- H.3, derived from A.2
   I.3:  ASCENDING(X(X'FIRST..X'LAST)) -- replacement in H.6
   I.4:  ASCENDING(X)                 -- I.3, and the Ada definitions of
                                      -- slices, 'FIRST, and 'LAST
```

The postconditions are just J.1, J.2, and J.3, so the procedure body is proven.

Note that throughout the proof, most of the deductions are either trivial or small-but-tricky: few are of middling difficulty.

That formulation of the pre- and post-conditions depended on the array to be sorted being "unique": that is, with no value in two positions (before, and therefore after). To allow duplicate values in the array, we would need to define `IS_PERM` differently, with a new function in virtual text used only in `IS_PERM`:

```
--:  function COUNT (E:ELEMENT) return  INTEGER is
--:     TO_DATE: INTEGER :=0;
--:  begin
--:     for I in INDEX loop
--:        if X(I) = E
--:        then TO_DATE := TO_DATE + 1;
--:        end if;
--:     end loop;
--:     return TO_DATE;
--:  end COUNT;

--: function IS_PERM(Y,Z:ARR) return BOOLEAN is
--: begin
--:     return (I in INDEX implies ( COUNT(X(I)) = COUNT(in X(I)) ));
--: end IS_PERM;
```

The `UNIQUE` function is no longer needed, and the precondition becomes null:

```
procedure SORT (X:in out ARR);
--| out IS_PERM(in X,X) and ASCENDING(X)
```

(There's a "scoping" issue there for the extended language and for the tools: `COUNT` directly uses the parameter "X" of `SORT`, and so should be "in the scope" of the parameter. That means some special ANNA rules on the placing of the definition of `COUNT`, and hence of `IS_PERM`.)

Direct Manipulation as a Basis for Constructing Graphical User Interfaces Coupled to Application Functions

JAN VAN DEN BOS
Department of Computer Science
Erasmus University Rotterdam
The Netherlands
E-mail: jvdb@cs.eur.nl

Abstract. Given a collection of application functions, a system is discussed that allows designers to construct the three parts of a graphical man-computer interface, t.w. layout (including presentation and graphical constraints), dynamic aspects (behavior or dialogue), and coupling to application functions, all by means of direct manipulation of graphical objects. In other words the methodology is based on interface-by-example, without the designer using a programming language. This system, called DIGIS, for Direct Interactive Generation of Interactive Systems, is presently in the design phase with prototyping being used to hone requirements and design, and to demonstrate preliminary results. Insofar as results from human factors (ergonomics) research are available, they will be applied to the design of the interactive system DIGIS itself, as well as to the target user interface. The implementation of DIGIS is done in the locally designed parallel object-oriented language Procol, a superset of C. It is built to be easily portable to existing windowing environments on workstations and personal computers, but the initial target is the X window system.

1. Introduction

Lots of applications are waiting for a usable human-computer interface. Such an interface would be based on sound visualization techniques and ergonomic principles, and be tailored to the intended group of users. If more than one user group would be involved this could result in several interfaces to the same collection of application functions. This many-to-one relation implies a loose coupling between the interface and the application functions.

The emerging visual and audio possibilities in interface technology have to be applied with care and intelligence. In the first instance a user interface should adhere to some existing use metaphor. It is a fallacy, however, to asssume that the full benefits of a new technology can be reaped when trying to squeeze everything into the harness of existing traditional methodology. New technologies have new possibilities and consequences: a car is not a coach, and a user interface is not a desktop.

Assuming that a collection of executable application functions exist, and assuming that their procedural interface can be described in an application interface model, we distinguish three major areas in the construction of a graphical user interface (GUI):

- **Static** or Layout part: graphical presentation, positioning, and constraints between presentation components (alignments, connectedness, ...);

- **Dynamic** or Behavioral part: determines the behavior of the presentation as a (partial) ordering in time in accordance with the user-executed dialogue;
- **Coupling** part: links input values to application functions, and maps possible application results to interface presentations.

Many so-called interface builders have been presented at scientific conferences, some are commercially available. For an overview see a more detailed paper on the project DIGIS [1]. However, few or none of these systems offer support for the Dynamic or the Coupling part: that usually has to be accomplished by writing pieces of code in a programming language.

It is sort of natural to expect that the construction of a GUI is carried out by visual means, that is by Direct Manipulation of graphical objects representing interaction tools, constraints, grouping, dialogue editors, etc. In practice however, several or all of these parts have to done by programming.

It is also accepted practice that the designer of a human-computer interface is a programmer. Translated to the automobile world, we would have a designer with a background of a car mechanic.

But an interface designer needs different qualities. For a GUI it is clearly required that he know about graphics design in the context of the limited facilities a bitmapped screen (often without colors) offers. He has to have a background in computers to determine feasible and optimal solutions. He must have some knowledge of cognitive ergonomics, or human factors as it is called in America. Finally, he ought to have an understanding of the general application area: architects require different interfaces from greengrocers. In other words, the software engineer who plays the role of interface designer must be more versatile than the software engineer building non-interactive applications. It is clear that special education in this area is badly needed.

2. DIGIS Description

The purpose of DIGIS is to provide a software engineering environment for user interface design, sometimes abbreviated as UIDE. This environment is itself an interactive and graphical system consisting of a graphical editor with tools tailored to the design of (graphical) target user interfaces (TUI). It draws on a toolkit with adaptable Interaction Tools (IT), and a library of application functions of which the procedural interfaces (AID = application interface description) are known. The graphical editor almost exclusively employs direct manipulation techniques for copying, moving, scaling, connecting, etc. graphical objects. DIGIS generates a TUI, and couples it via the AI, to the application functions (see Fig. 1 upper part). Subsequently, the generated TUI can function as input to DIGIS for modification, correction, refinement and extension (see Fig. 1 lower part).

The application procedures are assumed to be available. The interaction tool (IT) kit first has to be built. It is to contain at least a skeleton for each IT in a set

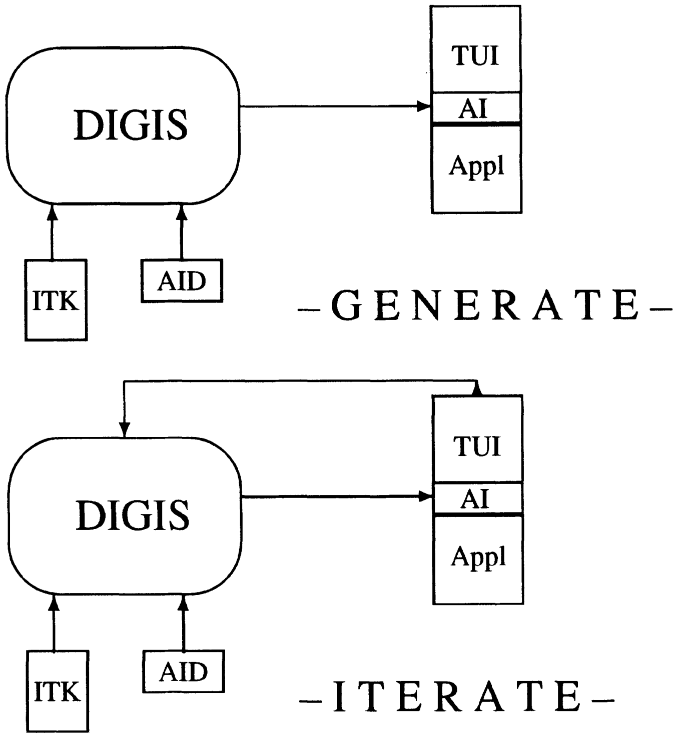


Fig. 1. The global architecture of DIGIS. At the top the generation, at bottom the iteration cycle. (ITK = input tool kit, AID = application interface description, TUI = target user interface, AI = application interface)

of complete and sufficient ITs for TUI design. An interaction tool is defined here as a method (a program component) that prompts for, reads and echoes user input, updates the state of the interaction tool, and that can be linked to a function. This function can be an interface function, an application function, or both.

Tool kits from existing interface standards, such as OpenLook and Motif, could be used as well. But rather than standardizing on a set of ITs with fixed presentation, it appears more user-friendly to provide the designer with means to modify and adapt the available ITs, and thus the IT kit.

The functional requirements set out for DIGIS are:

- Interaction tool kit (ITK)
- Adaptable basic ITK: presentation, interaction style
- Composite interaction tools built from basic interaction tools
- Multiple interaction tool kits (OpenLook, Motif)
- Placing, moving, sizing, and copying interaction tools

- Grouping interface components on screen
- Aids for constraints between interface presentation components
- Specification of dialogue: sequences, alternatives, repetitions
- All user input handled by user interface
- Interface output (possibly from application) handled by interface
- Application (esp. graphics) output rendered by application in application window(s) provided by DIGIS
- Saving and loading of intermediate TUI designs for possible modification
- Coupling to existing and new applications
- Interface coupled to application functions in C and other languages
- Methods to graphically link TUI to application and interface functions
  - parameterless coupling
  - mapping to *in* parameters in function parameter lists
  - mapping *out* parameters from functions to interface output
- Rapid prototyping by linking to stubs
- Accomodates various global interaction styles with local uniformity
- Portability to various window systems
- ALL BY DIRECT MANIPULATION

Essential is the last requirement. Where other interface builders use a mixture of programming and graphical manipulation, DIGIS stipulates that building the interface and coupling it to the application functions must be done by *direct manipulation*. This implies that with the exception of entering text (e.g. for labeling), all definition, manipulation and modification of the TUI being worked on should be done by the mouse operating on the graphical objects in the screen image of the TUI.

For pragmatic and philosopical reasons DIGIS is restricted to TUIs that do not allow the direct manipulation of *application objects*. To clarify this, take a screen oriented word processor. These programs often allow the selection of a piece of text that can subsequently be moved or copied to another place in the text. For such a TUI, DIGIS would support the (extended) selection of the text, but would not be able to move the text, the latter being the task of the application. These kinds of applications are invariably characterized by a spaghetti-type interweaving of interface and application.

But the goal of DIGIS is to maintain a strict separation between interface functions and application functions. DIGIS abstracts from the semantics of the application. This restriction precludes the construction of DIGIS in a bootstrap mode. Bootstrapping can only be done partially, as long as no direct manipulation of application objects, here DIGIS graphical objects, is concerned. Said in other words, DIGIS is not a visual programming environment, not even for a subset of applications, such as graphical ones.

A second restriction is the requirement of an existing IT kit. In practice this means that we have built our own IT kit by means of programming. Care has been taken to use parameterization, so that presentation and interaction style (what physical triggers, e.g. buttons, to use to trigger the IT) can be modified, when taking a copy of the IT from the tool kit for the TUI.

A third restriction is an unknown. It is hard to predict that one DIGIS system would cover all possible TUIs. For instance, (image) space plays an important role in the general tools that DIGIS provides to build a TUI. But many applications, such as real-time process control, deal with time in a rather precise manner.

This would have obvious results for the general tools that DIGIS had to provide in those cases. It would even have repercussions on the description of the application procedural interface, the AID. Data about the actual duration of (paths in) time-critical functions would have to be known. This would not only make the TUI computer-dependent, but in a distributed run-time processing environment, the TUI designer would also be forced to indicate in one way or another, which processor (given inhomogeneity) would have to be selected. Another potential problem could be a TUI for an information system. It seems that we can build such a TUI with confidence, but if the relation of the TUI with the data became too strong, an application *data* interface description might well be necessary.

At present we are constructing our third DIGIS prototype, analyzed and designed by means of the object modeling method OMT [4]. It it still restricted to the specification of the static part of a TUI. Presentations of ITs can be taken from the IT kit, and be positioned, copied, moved, and scaled by means of IT *shadow* objects (see [1]). ITs on the intermediate TUI can be grouped in Panels, which act like composite ITs. These Panels can also be added to the IT kit.

A problem here are the layout constraints, such as alignments (horizontal, vertical, diagonal, ...), fixed or relative dictances between ITs, etc. In principle, all this can be solved by letting the designer think in terms of (2D) coordinates. From an ergonomic point of view it appears better to offer the designer higher-level concepts, such as iron bars (for fixed distance), springs (for maintaining relative distance), hinges (for connections), and so forth.

The static part could definitely benefit from ergonomic rules, guiding the layout, information density, use of color and patterns, etc. One could think here of help files for consultation, or perhaps even better, automatic help in the form of suggestions. It is tempting to say that expert systems could help here.

The next problem to be solved is the dynamic part of the TUI. This deals with the behavior of the TUI in time, as a result of the dialogue between user and TUI. We call this the dialogue editor. In fact it already plays a role in the composition of ITs into a Panel, because composition in general implies a (sub) dialogue over the constituent ITs. This part appears to be harder to solve by direct manipulation than the static part. The reason is that in the static part we are solving graphical problems graphically. Specifying the dynamic part means using graphical means to define ordering of the dialogue components and the resulting difference in presentations, in time. In Fig. 2 a possible solution is suggested, but event-driven state diagrams or, equivalently, (augmented) regular expressions [3], could be used as well.

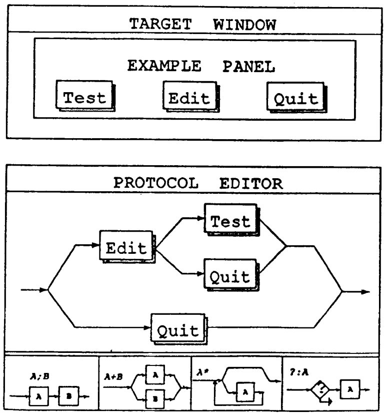


Fig. 2. A proposal for a dialogue editor for DIGIS. At the top the intended target user interface. In the middle the editor with an action graph of the dialogue. At the bottom the 4 basic primitives (sequence, selection, repetition, and guarded execution) from which an action graph can be composed, are shown, together with a linear notation.

Finally, ITs have to be linked to ITs or application functions. To be more precise, for coupling to application functions, the output or results of (composite) ITs have to be mapped to the parameter lists of the application function involved. The simplest case is where the activation of an IT triggers the function, without any (new) information being transferred. But in general, results have to be mapped to one or more parameters in a parameter list. This may even require some transformations, e.g. from string to numeric, or from integer to float, or assigning data to fields in a structure or message. It is conceivable that DIGIS offers a number of standard 'mappers' to do this, although strictly speaking, this should be part of the application. The most complex case is when the application function returns a result, important for the status of the TUI. This is commonly called semantic feedback. It immediately suggests that ITs are not only activated by a user, but also by the application.

An artist's impression on how to accomplish part of the job is presented in Fig. 3. It is assumed here that an AID is available, or has been made, so that the parameter list can be graphically and textually displayed (middle part of figure).

The example links a composite IT with one button (`Save`) and two text fields (`Dir` and `File`) to a function which saves the named file from a given directory. The designer specifies this by displaying first the application routine window. The coupling option of the Save button pops up a menu, with the option to link to application. He maps by pointing to this menu item and dragging an arrow point to the name of the routine (`SaveFile`). Subsequently, by similar methods he maps the state of the text fields to parameters `Dirname`, and `FileName`, resp. The success or failure of this operation (`Result`) is in this case not used: the designer

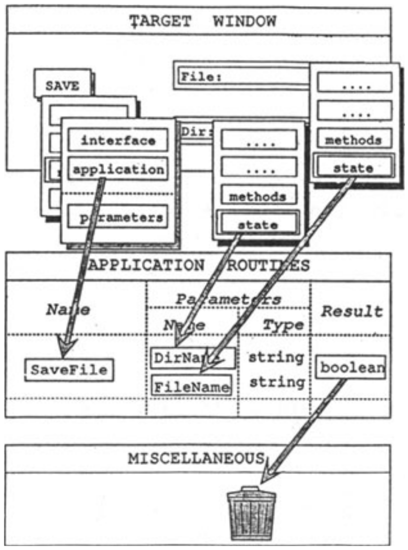


Fig. 3. Linking an Interaction Tool to an application function by means of pop-up menus and direct manipulation (pointing and connecting arrows). The middle part shows the procedural interface of the application function. The arrow to the bottom part indicates that the semantic feedback of the function is discarded.

indicates that it has to be discarded.

2.1. Ergonomics

All three parts could obviously benefit from (cognitive) ergonomics. The problem is that this discipline has not put forward a single comprehensive theory from which a body of rules and guidelines could be drawn. Partial results on models for task solution have been presented, but empirical results are equivocal and inconclusive. So lots of rules seem little better than common sense, experience, or folk wisdom. Of course, graphics design is an established body of knowledge, daily practiced on e.g. newspaper layout. But it deals only with the static part. It remains to be seen if a layout meant to be interactive, should be similar to a static layout only meant for viewing or reading. Yet I still consider this our best source for guiding rules in interface design.

In DIGIS, ergonomics crops up at two places. First of course in the target user interface onder construction. But we also need ergonomic guidance for the design of DIGIS itself: what sort of techniques, and how, should be provided to the designer? Should they vary with designers?

Ideally a body of rules should be put in an expert system. But the state of knowledge being what it is, one would perhaps couple two areas with questionable results: artificial intelligence and ergonomics.

2.2. Implementation

The IT kit, as well as all other parts and techniques of DIGIS are programmed in the object-oriented language (OOL) Procol [2]. It is a language we designed as a superset of C. Different from most OOLs, all its objects execute in parallel. Furthermore, it uses delegation (more tailored to a distributive processing environment) rather than inheritance. It also offers (one-way) constraints and, very appropriate for input tools, Procol's objects all contain an explicit message protocol, in which access ordering and clientele may be specified. This protocol maps very closely to a dialogue. Because Procol uses C as its host language, it is fully compatible with existing C libraries, in particular the X windows library, which is our initial platform.

The DIGIS implementation uses a well-defined and localized channel to X, so that dependencies on X are minimized. This is because the intention is to make DIGIS portable to other systems, such as Microsoft's Windows.

3. Conclusion

We have discussed principles of graphical user interface construction, and their embedding in an environment for user interface design, DIGIS. Three important subsystems have been distinguished: the static or layout part, the dynamic or dialogue part, and the coupling to application part. In DIGIS all these parts are specified without programming. Instead it uses direct manipulation of graphical objects on the screen. A prototype of the first subsystem is available, as well as preliminary designs of the other subsystems.

Acknowledgements

This project is supported by STW, the Foundation for Technical Research in the Netherlands. I also thank my collaborators Hans de Bruin and Peter Bouwman.

References

1. Van den Bos, J. and Laffra, C.: 1990, 'Project DIGIS: Building Interactive Applications by Direct Manipulation', *Comp.Graphics Forum*, **9**, 181-193
2. Van den Bos, J. and Laffra, C.: 1991, 'PROCOL, A concurrent object-oriented language with protocols, delegation and constraints', *Acta Informatica*, **28**, 511-538
3. Van den Bos, J.: 1988, 'ABSTRACT INTERACTION TOOLS: A Language for User Interface Management Systems', *ACM-TOPLAS (Transactions on Programming Languages and Systems)*, **10, 2**, 215-247.
4. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: 1991, *Object-oriented modeling and design*, Prentice-Hall, New York

Visualization of Volumetric Medical Image Data

K.J. ZUIDERVELD AND M.A. VIERGEVER
Computer Vision Research Group,
University Hospital Utrecht, room E.02.222,
Heidelberglaan 100, 3584 CX Utrecht, The Netherlands
E-mail: karel@cv.ruu.nl, max@cv.ruu.nl

Abstract. This paper presents on the state-of-the-art of volume visualization methods that are tuned to visualization of medical datasets. After topics as sampling theory, data preparation, and shading, various visualization strategies are explained with an emphasis on algorithms that do not rely on special hardware. Special attention is paid to strategies that improve image generation speed, while techniques for improvement of image quality are also discussed. Finally, trends in this area of research are identified.

1. Introduction

Digital medical imaging modalities, *e.g.*, Computed Tomography (CT) and Magnetic Resonance Imaging (MRI), are important means of providing information about anatomical structures. These modalities produce 3D image data sets, usually consisting of between 8 and 300 cross-sectional slices, where each slice has a resolution of 256^2 or 512^2 pixels. A good understanding of the 3D anatomy is of paramount importance for diagnosis, surgery planning, and therapy planning. Consequently, it is essential to extract the relevant information contained in the vast amount of 3D image data and present it to the clinician in such a way that complex anatomical structures and their interrelationships can be better comprehended.

It has appeared virtually impossible to mentally reconstruct a 3D picture that is consistent with the large number of cross-sectional images. This is true in particular for highly pathological studies that lack a frame of reference as is provided by knowledge of the human anatomy in non-pathological cases. Volume visualization has already proven useful as an aid in diagnostics and therapy planning, especially in craniofacial surgery and orthopaedics [1; 2].

3D datasets represent a wealth of image features of which only a minor fraction is of real interest. In addition, clinicians want to have the ability to visualize specific tissues or organs, usually in relation to their surrounding structures. An extremely crucial step in the process of visualization is, therefore, a segmentation[1] of the volume data which identifies objects of interest.

There is a growing tendency to combine volume data from different modalities, *e.g.*, anatomical information (obtained with CT and MRI) with images providing functional information (*e.g.*, Positron Emission Tomography (PET) and Single

[1] Segmentation is the division of an image into meaningful entities. In the strict sense of the term segmentation, the meaning of the entities is not explicit until a classification (or labelling) has been performed. In this paper we interpret segmentation in the wide sense, *i.e.*, with inclusion of the classification step.

Positron Emission Computer Tomography (SPECT) data). Here, even more than for single modality images, the problem of reconstructing mentally a 3D picture comprising the information provided by the various modalities occurs. Integrated 3D display of multimodality images, depicting simultaneously aspects of anatomy and function, is called for. An important prerequisite for multimodality imaging is the availability of methods that reliably match the images acquired from different sources. We therefore find that work on volume visualization heavily relies on developments in the areas of segmentation and multimodality imaging.

Medical volume visualization is an interdisciplinary field; it uses skills and experience from areas as image processing, computer graphics, numerical mathematics, visual psychophysics, various branches of medicine, and ergonomics. It would be beyond the scope of this paper to extensively survey the role of each of these disciplines. We have chosen to concentrate on the methodology of visualization and virtually ignore important topics as user interfaces, parallel hardware and clinical applications.

In this paper, we give a state-of-the-art report of volume visualization methods for medical image data. We first explicitly address various topics that we consider important for obtaining a general understanding of the main problems in volume visualization, namely data preparation, sampling aspects and shading. We then survey currently used methods for medical volume visualization, with an emphasis on algorithms that do not rely on special hardware. Next, attention is paid to techniques that improve image generation speed and to strategies for improvements of image quality. Finally, some trends in the area of medical volume visualization are identified.

2. Basic topics

This section provides some information on data preparation, segmentation, sampling aspects, and shading. We feel that an explicit discussion of these topics is helpful to get a better understanding of the problems that various visualization methods try to deal with.

2.1. Data preparation and segmentation

An important step in the visualization pipeline is data preparation. This includes

- 3D patient motion correction;
- multimodality matching, *e.g.*, anatomical information (CT, MRI) with functional information (SPECT, PET, EEG);
- image correction, *e.g.*, for inhomogeneous magnetic fields in MRI data;
- interpolation;
- contrast enhancement.

The prepared volume data contains millions of volume elements (voxels), of which only a part is really of interest; a reliable segmentation step that identifies and classifies these interesting voxels is crucial. Much work still has to be done regarding the development of satisfactory (and, given the size of the dataset, fast) algorithms for automatic 3D object recognition and tissue classification.

In one of the many application areas, craniofacial reconstruction, the main tissue of interest is bone. Here one can take advantage of the relationship between tissue density and image greylevel in CT. It can safely be assumed that all density values above a certain greyvalue correspond to bone, so simple thresholding suffices to segment out bone voxels in the dataset (however, due to partial volume effects, some voxels containing bone will not be recognized). Segmentation of CT data into the reasonably wide classes fat, soft tissue and bone using intensity values only is adequate for many purposes [3]. It is therefore not surprising that the vast majority of clinical (and theoretical) papers on medical volume visualization bear upon CT data and are restricted to visualizing bone (or sometimes bone and soft tissue).

If automatic segmentation is not possible, interesting image features must be manually entered. This is usually done in an interactive session on a slice-by-slice basis. Since a patient study can contain hundreds of relevant contours, complete manual contouring is extremely cumbersome and thus expensive. There is a clear need for better editing tools to reduce this burden [4].

The segmentation step results in a model of the original data. Various models have been proposed, such as a series of contours, a volume of binary voxels, or an encoded version of the latter (*e.g.*, using octrees). These possibilities will be discussed in more detail in the sequel. Using a model has the advantage that one obtains a large amount of data reduction; on the other hand do errors in the model, *e.g.*, incorrect tissue classifications, likely lead to artifacts.

In clinical studies, the separation between individual slices is usually greater than the pixel size of the slices (especially in studies obtained using standard s-canning protocols not aimed at 3D reconstruction). These datasets suffer from irreversible undersampling, leading to severe aliasing artifacts that can not be removed by greyvalue interpolation. In such cases, acceptable results can be obtained using a shape-based interpolation algorithm [5], that marks interesting voxels of the segmented volume using the shapes of the cross-sectional boundaries from adjacent slices.

Visualization software initially only relied on models, mainly because there was not enough computational power and memory available to handle large datasets. More recent algorithms tend to use the original volume data, but may, in addition, also use coded versions thereof to exclude irrelevant image data during the volume visualization.

2.2. Sampling aspects

In order to recognize potential problems of volume visualization, a few issues related to sampling theory must be addressed.

- The input volume consists of samples at regular spatial (and possibly temporal) intervals. In many visualization algorithms, data has to be interpolated, often thought of as the process of "filling in" points between original data. Actually, interpolation is reconstructing the continuous function from its discrete sample points, after which intermediate points are obtained by resampling the continuous function. Assuming the volume dataset has been properly sampled, reconstruction of the continuous function (the original 3D scene) is possible by convolving the samples with a large kernel containing a 3D sinc function. In practice, computing the original 3D continuous function in this way is just too expensive, given the size of the kernel and the dataset. A more feasible approach is to combine the reconstruction of the continuous function with the resampling step, which can be done using any interpolation algorithm. Nearest neighbour and trilinear interpolation is used almost exclusively here. However, these algorithms have been found to show rather poor spectral behaviour [6]. Cubic splines have superior spectral characteristics, but are computationally more expensive [7].
- Modalities as CT and MRI have a limited resolution; the obtained volume datasets are therefore band-limited (filtered) representations of the original 3D scene that is not band-limited. Problems can be expected with the visualization of small or thin objects (*e.g.*, tissue boundaries); a good example of this phenomenon is the *partial volume effect* associated with CT imaging, which is essentially the result of the low-pass bandfiltering associated with the tomographic reconstruction.
- In volume rendering, we are essentially calculating a band-limited 2D projection of a 3D scene. In ray casting algorithms, which will be discussed in the next section, the number of rays should be chosen high enough to avoid sampling artifacts (aliasing). However, the number of rays is usually fixed (one per output pixel) which implies that we depend on 3D band-limiting to avoid aliasing in the 2D projection. If this provides for insufficient sampling, artifacts can be avoided either by slightly blurring the 3D dataset or by increasing the number of rays cast (supersampling).

2.3. Shading

Volume visualization relies on shading techniques to produce realistic images. These techniques model the process of light absorption, reflection, and transmission along surfaces; this process largely depends on the properties of the surface material.

Although usually not explicitly mentioned, a few topics should be addressed when shading calculations are performed at arbitrary positions in the volume.

2.3.1. Surface classification

We distinguish two approaches to determine the location of object surfaces, depending on the availability of a classification volume:

– *Classify surface based on greylevel data*
  With this approach, the (interpolated) sample point greyvalue is checked against ranges that are typical for specific tissues. If the greyvalue falls outside these ranges, the sample point can be skipped. The binary decision whether or not a voxel belongs to a limited set of tissues may lead to severe artifacts. This can be circumvented by allowing probabilistic classifications that set the sample point opacity to values between 0 and 1 [8].
– *Classify surface based on classification data*
  With this approach, the presence of a surface is determined from the classification data, which usually compromises a volume with the same dimensions as the greylevel data. Therefore, the classification volume also consists of regularly spaced samples. Strictly speaking, one should interpolate between classification data from voxels adjacent to the spatial point of interest to obtain the required information, which results in probabilistic classifications even when the original classification data is binary.
  Most volume visualization approaches settle for nearest neighbour interpolation by — either explicitly or, more often, implicitly — using the classification of the voxel nearest to the sample point.

From the point of signal theory, the first approach is preferable if there is a clear relationship between greyvalues and tissues (the classification step represents a non-linear transformation that has undesired spectral properties which cause aliasing artifacts in the resulting image). However, the implicit assumption of proper sampling is often violated; in such cases, shaped based interpolation of the (binary) classification data offers interesting possibilities.

It is interesting to note that both approaches support the notion of probabilistic classification values (*e.g.*, a voxel contains 40% bone); this value can be used to modify the opacity of the sample point.

2.3.2. Surface orientation

Light models that are subsequently used in the shading process often require the value of the surface normal at the sample position. The following methods have been proposed to determine the surface normal:

– *Weighted surface gradient*
  This method is used in implementations where memory usage should be kept to a minimum; an approximation of the surface normal is directly calculated from the classification data. Owing to the binary nature of the segmented data, only a limited number of normal directions are possible. This facilitates fast implementations (*e.g.*, using look-up tables), but also leads to visible artifacts as intensity contouring [9].

Thanks to recent hardware developments, memory usage has become less critical, which implies that this method is gradually superseded by others. However, specialized hardware architectures for medical volume visualization still rely on this method [10; 11].

- *Normalized greylevel gradient*
  A currently popular method is the use of the normalized greylevel gradient as the surface normal [12]. The gradient is calculated either from the greylevels of its 6 first order neighbours or from the greylevel data in a second order (3×3×3) neighbourhood of the point of interest. Normalization of the resulting gradient value then yields the surface normal. This method does not suffer from contouring artifacts due to its high dynamic range.
- *Spatially adaptive gradient*
  Pommert et al. [13] report that for thin objects even a first order neighbourhood may be too large. They present an algorithm that adapts to the thickness of the object and demonstrate that the resulting method significantly improves the accuracy of the computed surface normals. Essentially, they use a simple criterion based on the local frequency contents to choose from two differently sized filters. This method can be generalized to a multiscale approach based on Gaussian filtering to derive the local gradient value [14]. Owing to the huge computational requirements, this multiscale approach is at present not suited for practical applications.

While these methods calculate the surface normal at the grid points, sample points may not coincide with these positions and interpolation issues should be explicitly addressed again.

Surface normal calculations represent a heavy computational burden, hence most implementations perform these calculations during a preprocessing step. An interesting issue is storage demands: Since "dumb" storage would require too much memory (using 3 floats per voxel requires about 200 MBytes for a 256^3 dataset), data reduction must be used. One way to achieve this is by only storing gradient values for voxels that lie on the surfaces of segmented structures. Another option is vector discretization where for each vector two components are quantized to a low number of bits whereas of the third component only the sign is stored, taking into account that the surface normal is an unit vector.

2.3.3. Light reflection

Given the presence of the surface and its orientation, calculation of the light that is reflected from the surface is straightforward. Since photorealism is not required, a simple light reflection model is adequate for most visualization purposes. The customary model is that of Phong [15], which separates the reflected light into three components:

(i) an *ambient* component, representing the contribution from various distributed light sources that are too difficult to model. This is usually a constant term which only depends on the ambient reflection coefficient of the surface and the colour of the ambient light;

(ii) a *diffuse* component, representing reflection that radiates equally in all directions. The amount of diffuse reflection only depends on the flux of the light (which is obtained by calculating the inproduct of surface normal with the direction of the light), it is independent of the viewer position.

(iii) a *specular* component, representing the "shiny" reflections which are typical for metal-like surfaces. This contribution depends on the surface orientation, the position of the light source(s) and the viewer position. Modelling of surface shininess is a quite expensive operation (and therefore implemented for choice using look-up tables).

Surface transparency is not supported by the Phong light model, but can be included by a straightforward extension.

If the specular contribution is omitted, fast implementations are feasible since colours at the voxel positions are then independent of the viewer position. The light reflected by each voxel of the volume can be precalculated provided the position of the light source(s) is unchanged, which gives to fast image generation speeds but relatively poor image quality (the observer is "moving" around the object). In general, better image quality can be obtained against higher computational costs.

We notice, in passing, that dissimilarities between images of various institutions usually do not originate from fundamental disparities in the visualization approaches, but are caused by choosing different light models.

3. Visualization methods

During the second half of the eighties, numerous visualization methods have been developed and applied to clinical problems, with varying success. Visualization strategies that focus on the display of surfaces can be divided into two categories: Surface rendering and volume rendering (*e.g.*, [16]). Surface rendering is performed by describing the objects as well-defined, infinitesimally thin geometrical primitives and using shading techniques from the field of computer graphics to calculate the amount of reflected light from each surface element. In contrast, volume rendering uses volume elements as its basic primitives; we distinguish between binary class volume rendering, which uses binary classification data and graded class volume rendering, which incorporates probabilistic classification data in the visualization algorithm [17].

Other well-known visualization algorithms are multiplanar reformatting and transmission-oriented methods; the latter is often used for 3D display of noisy volume datasets.

3.1. Multiplanar reformatting

If we define volume visualization as interactive exploration of volume data, an important technique is multiplanar reformatting which resamples the volume data (usually along a plane) and displays the result in a separate image. In its simplest form, a single orthogonal section through the volume is displayed. More advanced implementations simultaneously show multiple orthogonal sections (coronal, sagittal, lateral), are able to display greyvalues along planes of arbitrary directions (oblique sectioning), or even along curved lines that can be interactively drawn [18]. Thanks to the simplicity of these techniques, current workstation technology facilitates real-time display and manipulation of cutplanes in volume data.

3.2. Triangulation based on surface contours

In triangulation methods, the model of the volume consists of a set of contours representing surfaces of interest; the basic idea is to fit a skin over the individual contours using triangular surface elements. Various algorithms for triangulation have been published [19; 20]; they generally try to minimize a cost function (*e.g.*, total triangular surface, lateral length, volume). The triangulation is followed by rendering the obtained primitives using a conventional graphics pipeline; the graphical hardware of modern workstations is fast enough to achieve real-time visualization of surfaces with moderate complexity.

Organs can have very complex topological structures, which makes successful triangulation dependent on substantial help from experienced users. Recently, an algorithm was published which calculates Reeb graphs that describe the topological relation between contours in successive cross-sections [21]; initial results indicate that, aided by some a-priori information, this algorithm works better than methods based on human interaction.

In general, triangulation methods yield coarse images. Their use is therefore limited to applications for which high detail is irrelevant.

3.3. Iso-value contour display

Iso-value contour methods are especially tailored towards the calculation and display of iso-value surfaces in volume data; they are computationally intensive, but yield high quality images.

One of the first published algorithms, called *marching cubes*, converts the original dataset into a very fine polygon mesh, suitable for rendering by graphic accelerators [22]. Surface topologies are determined within logical cubes that are defined by eight voxels, four each from two adjacent slices. The positions of the polygon surface(s) within this cube are then determined by trilinear interpolation of greyvalues.

Recently, an improved triangulation method was presented by Wallin [23]; it explicitly takes into account surface coherence by using a temporary list in which the polygon edges are stored. Subsequently, the edge list is traversed to produce surfaces of ordered polygons.

These algorithms typically generate numerous tiny polygons that occupy only a small area of the final image. Although it is possible to decrease the number of triangles by a post-processing step, it is often not performed because of resulting image degradation. The sheer number of triangles represents a serious burden on memory bandwidth and processing power of machines; only special purpose graphics hardware is able to provide acceptable image generation times.

In contrast, the *dividing cubes* algorithm [24] circumvents this problem by choosing a surface element about equal to the pixel size, which results in a cloud of points. Each voxel of the dataset is subdivided into small cubes with a size that matches the pixel size of the output image; the densities at the corners of each cube are calculated from the eight corresponding voxel vertices by trilinear interpolation. Each cube is then classified as being outside the surface, inside the surface, or intersecting the surface. For those cubes that lie on the surface, the gradient vector is calculated, shading is performed and the resulting colour is projected onto the viewing plane.

These surface rendering algorithms are not popular although they result in high-quality display and do not require previous volume segmentation. One of the main drawbacks is their inability to display non-iso-value surfaces, which are often found in MRI and SPECT datasets. However, these algorithms are well suited for specific clinical applications [25; 26]; the most probable reasons for their unpopularity is their obscurity and the slow speed on non-graphics hardware.

3.4. Binary class volume rendering

Binary class volume rendering methods are currently the most popular in clinical applications, because they are fairly fast and deliver good image quality [27]. In a preprocessing step, relevant object surfaces are marked in a binary volume; this segmentation is sometimes done during the image generation step by applying greyvalue thresholding.

Shading calculations are only required for voxels on surface boundaries; local surface orientation is usually obtained by applying a 3D gradient operator on the segmentation data [28] or on the original volume data [29].

Close inspection of actual implementations reveals some interesting differences. Globally, two ray casting methods can be distinguished: Given the scene, project voxels onto the screen (*forward mapping* or *object order* algorithms) and given the screen, cast rays into the scene (*backward mapping* or *image order* algorithms).

With backward mapping, values of the original dataset rarely fall exactly along the ray. In order to obtain sample values along the ray, approximations are calculated by interpolation of the volume data during the actual ray casting step [8].

Forward mapping algorithms directly project the volume data on the image plane. In order to avoid problems with points not mapping onto grid positions of the resulting image, the complete 3D volume data must be geometrically transformed to match the desired resolution of the output image. Calculation of a different viewpoint thus implies a geometrical transformation of the complete dataset, a time-consuming operation [30; 12].

These techniques suffer from the problem of having to make binary classifications: A voxel contains either a surface or it does not. As a result, these methods can exhibit spurious surfaces (false positives) or erroneous holes in surfaces (false negatives).

3.5. Graded class volume rendering

Graded class volume rendering techniques incorporate the concept of probabilistic classification into the visualization by extending the used light model with transparency. Along each ray cast into the scene, a vector of sample colours and opacities is determined. A fully opaque background is draped behind the scene, after which the colours and opacities along the ray are merged with each other by compositing (often referred to as blending [30; 31]). The opacity of individual voxels should be directly related to the probability of the presence of interesting tissue. Consequently, good tissue classification algorithms are still required in volumetric compositing; however, they do not need to provide perfect binary classifications.

In CT datasets, probabilistic classification of fat, soft tissue, and bone is feasible using only the absolute greylevels of voxels; examples of such classifications have been published by various groups [32; 33]. Since users are mainly interested in tissue boundaries, the gradient amplitude is often used to reduce the opacity of voxels in homogeneous parts of the volume that can be characterized by low gradient values [8].

Volumetric compositing methods are rather demanding as concerns computation time; reasonable image generation speeds can only be obtained by special-purpose hardware [34] and/or programming tricks [35; 36; 37]. Although the generated images are of high quality, only a few reports have been made on clinical applications [38; 3].

3.6. Transmission-oriented methods

The use of algorithms for surface display causes problems when the original data does not contain clear surface boundaries, as in SPECT and ultrasound imaging. In such cases, transmission-oriented display is often considered a useful alternative, of which X-ray projection and brightest voxel display are the most frequently used techniques.

The X-ray projection method simulates X-ray absorption by assigning an absorption coefficient to each voxel; subsequently, the total absorption along rays is calculated. This method can be used to calculate simulated X-ray images from

CT volume data.

Brightest voxel display (also referred to as Maximum Intensity Projection, MIP for short) displays the maximum value encountered along each ray cast into the volume. This technique is currently popular for the display of Magnetic Resonance Angiography (MRA) volume data, where each voxel value represents the amount of local blood flow. Unfortunately, MIP presents an inconsistent spatial perspective to the user; high intensity values obscure other features of interest and lead to irregular depth positions of the projected voxels. There is a clear need for modified MIP algorithms; an example is the Closest Vessel Projection which limits the projection range to the lumen of the detected vessel closest to the projection plane [39]. Unfortunately, such algorithms tend to rely on preprocessing steps that mark interesting voxels — a non-trivial classification problem on account of the significant amount of noise in the volume data.

Transmission-oriented methods suffer from the same problem as any ray casting method: The required geometrical transformation between object and screen space implies resampling of the volume data. For reasons of speed, the resampling step is often restricted to nearest neighbour interpolation, which can give disturbing aliasing artifacts. Another disadvantage of these methods is their computational burden. Since in principle each voxel of the volume data contributes to the output image, the ray casting process is time consuming in contrast with surface display, where techniques such as octree encoding and bounding volumes can be applied to skip irrelevant subvolumes. In some applications, however, preprocessing algorithms are capable of reducing the amount of calculations by identifying relevant structures. An example is the use of greylevel thresholds to identify potential blood vessels in MRA volume data.

For a good appraisal of the 3D structure, it is essential to dynamically change the viewpoint (*e.g.*, by continuously rotating the 3D object in front of the observer) [40].

3.7. Discussion

Although extensive discussions on the relative merits and disadvantages of individual techniques have been published (*e.g.*, [41]), one of the most important lessons learned is that, in general, there is no "best" method. The choice of a suitable approach strongly depends, among others, on the specific application, the quality of the volume dataset, and the available hardware.

Rendering techniques using binary classification data are often referred to as more practical, since they are relatively easy to implement and have limited memory requirements. Proponents of volumetric compositing point out that relatively inexpensive parallel processors will soon be capable of "real-time" visualization. Volume rendering also benefits from its greater functionality since the user has the opportunity to tune the algorithm to produce surface-rendered views as needed. It also requires less stringent preprocessing compared with surface rendering where the quality of a rendition critically depends on correct binary characterization of surfaces.

It is again important to note that most methods make the implicit assumption that the volume data was obtained using "proper" sampling; unfortunately, medical datasets are often severely undersampled along the scan direction. As a result, the dataset cannot be reliably interpolated, which leads to classification errors and shading artifacts. This explains the interest of the visualization community in shape based interpolation.

Also, there is a continuing interest in better acquisition methods. Recently, spiral scanning CT [42] has been introduced which has great potential for high-quality 3D imaging. New acquisition methods as 3D Echography pose new challenges with respect to 3D volume visualization and will likely lead to new visualization strategies [40; 43].

4. Improving image generation speed and quality

Given the basic visualization strategies, how can we optimize their performance with respect to speed and quality? With the contemporary hardware, we can not have the best of both worlds — yet. We will discuss how either of these contrary objectives can be achieved.

4.1. Improving image generation speed

Strategies for speeding up image generation are based on precomputation of parameters before the actual volumetric composition, taking advantage of coherence in the dataset or the resulting image (octree, adaptive sampling), or improving speed at the cost of image quality. Although we have attempted to categorize the methods, it should be noted that practical implementations tend to rely on several acceleration strategies, which precludes any strict categorization.

4.1.1. Adaptive sampling

Adaptive sampling, a method with roots in the field of computer graphics, was first applied to volumetric compositing by Levoy [35]. It uses the presence of coherence in the output image. A coarse image is produced by casting a sparse grid of rays into the volume, calculating the resulting ray colours, and bilinearly interpolating the obtained results. In regions of high image complexity the number of rays is increased; this can be done by a recursive subdivision based on local colour differences.

The method offers a good response time. Crude images can be obtained in a relatively short time interval, followed by gradually improving images at regular intervals as long the user does not change the rendering parameters. Since adaptive sampling involves some overhead, the total rendering time increases slightly.

4.1.2. Spatial encoding

Many datasets contain coherent regions of uninteresting voxels. Several techniques have been developed for encoding coherence in volume data. Levoy [36] introduced a hierarchical enumeration scheme using a pyramid of binary volumes. The basic idea is that during the volumetric compositing an interpolated sample is not interesting if all the 8 surrounding voxels have zero opacities. A volume is created where each voxel contains a binary flag indicating the presence of tissue; from this volume a hierarchical binary pyramid is constructed marking the presence of non-zero opacities at various resolutions of the volume.

During ray casting, the pyramid is also traversed; at entering a cell its value is tested. A value of zero implies total transparency, upon which the ray is advanced to the next cell on the same level, thus saving time by pruning away uninteresting regions of space. It must be noted that the pyramid, since it depends on opacities rather than on the original data, must be recomputed whenever the user adjusts opacity assignments.

A multiresolution approach has also been applied to iso-surface display. Wilhelms and Van Gelder [44] store local minima and maxima at various resolutions in order to be able to skip cells in case the desired iso-value falls outside these extrema.

The above techniques trade a longer preprocessing step for faster generation times; they do not cause any degradation in image quality.

Laur and Hanrahan [45] follow another strategy: They use a pyramidal volume representation to store average opacity values as well as the estimated errors. This approach is feasible thanks to the use of a splatting algorithm [37]; the required reconstruction filter ("footprint") is calculated for each level of the pyramid. This algorithm combines successive refinement with spatial encoding; when large errors are acceptable, rendering is performed at a low resolution which facilitates interactive speeds.

Udupa and Odhner [46] assume a binary segmentation volume which they then encode as an indexed list of voxels that lie on the surface of the object — voxels that are entirely inside the object are neglected. The semi-boundary list contains, besides the surface normals and voxel coordinates, an encoded representation of the visibility of each cell side with respect to adjacent cells. During image reconstruction, a simple table look-up is capable of determining potentially visible surface cells; since these comprise only a small fraction, this approach leads to significant computational savings. The final decision on visibility of the potentially visible voxels is then made by projecting the voxels back-to-front, after which shading is performed. The authors claim that, besides a great reduction in storage requirements and rendering time, this encoding strategy is very suitable for interactive manipulations as cut away views, mirror reflection and segmental movement.

4.1.3. Adaptive ray termination

Volumetric compositing techniques usually assume a back-to-front evaluation order [30]. A mathematically equivalent formulation facilitates front-to-back (FTB) ray casting. With FTB techniques, *rays* are assigned a colour and an opacity. Once a ray strikes an opaque object or has progressed a sufficient distance through a semi-transparent object, the opacity accumulates to a level that the colour of the ray stabilizes. Significant accelerations are obtained if the ray casting is terminated as soon its opacity reaches a user defined threshold level. Low thresholds reduce rendering time (typically a factor of 3 with an opacity threshold of 0.95), while higher values increase image quality, but at higher costs [36].

4.1.4. Precalculation of intermediate data

Depending on the specific algorithm used, several stages in the rendering process can be calculated and stored for subsequent use during the rendering step. An example of using precomputed data is Levoy's backward mapping method [8]. In his implementation, shading of each voxel is performed before ray casting and the result stored in an intermediate 3D dataset. The actual image generation step then only consists of trilinear interpolations between voxel colours followed by a compositing step. This approach does not have any effect on image quality when shading is non-specular (independent of viewing direction); incorrect highlights are introduced when specular shading is used. Levoy reports that observers are seldom troubled by these effects. Another example of this technique can be found in [30].

Foley et al. [47] present a method that is capable of approximating ray-traced volumetric images in less than one second per image. The speed is obtained by interpolating projection images that were calculated during a preprocessing step. Since this method is not able to correctly handle occlusions, the interpolant contains severe artifacts; however, for some applications the high image generation speeds may justify the errors.

A method by Gudmundsson and Randén [48] uses a list of surface points to take advantage of coherence between successive projections with small rotation angles, where the vast majority of surface points remains visible from one projection to the next. If surface points have gone out of hiding (have become visible), a few new rays have to be cast to calculate the corresponding output values; other values are obtained by appropriate geometrical transformations.

Precalculation of data usually introduces small artifacts, of which incorrect shading is the most frequent.

4.1.5. Simplification of visualization algorithms

Since the individual components used in the light model take different times to compute, major speedups can be obtained by neglecting the specular component or even the diffuse component. When enough memory for storing intermediate results is present, simplified shading can be implemented with successive refinement. Since the compositing step has to be repeated here, this method trades a faster response time against total computational costs.

Speed improvements can also be obtained by using nearest neighbourhood interpolation instead of more expensive interpolation methods. As already indicated, this leads to aliasing artifacts.

The mostly used methods for speed improvement are implementation "hacks", prominent amongst which are the use of look-up tables for shading and using integer calculations. They are usually tailored toward specific machine architectures, taking the available memory and cost of floating point operations into account.

4.2. Improving image quality

Evaluation of image quality is difficult. A "gold standard" is usually not available, in contrast with computer graphics applications where photorealism is the ultimate goal. A possible method to evaluate image quality is to perform 3D data acquisition on a phantom or to simulate the acquisition process and compare the rendered image with the expected result. Pommert et al. [49; 13; 50] use the latter approach to evaluate the image quality of various volume visualization algorithms.

Another approach to judge image quality is an evaluation of their ability to communicate essential information to the user. Special effects like shadows and perspective projection ("artistic devices", see [51]) greatly enhance the depth perception of the user but are not trivial to implement, as will be explained in this section.

We will discuss the main techniques for image quality improvements, again using a categorization into five strategies.

4.2.1. Higher order interpolation

The frequently used trilinear interpolation is a poor reconstruction filter, since it has only moderate frequency characteristics. A survey of various interpolation algorithms [7] revealed that the desired sinc interpolation kernel is best approximated by cubic convolution interpolation. This will generally lead to sharper and crisper images, but is in its generic 3D form computationally about 40 times more expensive than simple trilinear interpolation.

4.2.2. Supersampling

There are essentially two variations, *viz.* supersampling of the input dataset and supersampling during the image reconstruction.

The basic idea behind supersampling of the volume data is to interpolate additional samples between the acquired ones. This increases the effective sampling rate of the dataset, so the accuracy of the rendering increases. Practice shows that the improved quality hardly justifies the considerable amount of additional processing. Interpolation of original data does not add new information, but improves the spectral behaviour of the data (resulting in less aliasing) [8].

Another option is to apply supersampling during the compositing step by casting additional rays into the dataset (more than one ray per output pixel); scene pixels are determined by filtering the colours of different rays. This technique is widespread in computer graphics (see [52]). Adaptive sampling techniques as discussed in Section 4.1.1 can be easily extended to allow supersampling depending on image complexity.

The use of adaptive (super) sampling during the image generation step has been reported to give significant better image quality while keeping the additional computational costs low [35]; note that supersampling only reduces aliasing effects present in the output image and does not add new information.

The best way to enhance the intrinsic resolution of the output image of course remains to improve the spectral characteristics of the original volume dataset by refining the data acquisition methods.

4.2.3. Shadows

In real life, shadowing is an important cue to get insight in the structure of complex 3D objects. The addition of shadows therefore represents an important quality improvement.

When ray casting algorithms are used, the introduction of shadows is possible with a two-pass ray casting approach. During the first pass, ray casting is performed for each light source that radiates the volume data. Attenuation of light strength along an illumination ray is very similar to accumulation of opacity along a viewing ray, but the attenuation of the light with increasing distance to the light source should also be considered. Local illumination strengths can be stored in a 3D light buffer, after which, during the second ray casting pass, conventional shading based on these values is performed.

Since movie sequences are usually made with fixed light sources and a moving observer, the 3D light strength buffer is computationally cheap but very memory hungry. Examples of this technique are reported by Van der Voort [53] and Levoy [54]. A qualitatively better, but far more time consuming technique would move the object with respect to light sources, thereby providing more visual clues to observers.

A complication with this technique is that surfaces in volumetric data have the tendency to partially shadow themselves, introducing distracting aliasing effects. Levoy [55] proposed a simple solution by translating the 3D light strength buffer a few voxels away from the light source; this reduces distracting artifacts, but also decreases shadowing accuracy within small objects.

A different approach for providing shadows is the use of global illumination models that attempt to take into account the interchange of light between all surfaces in the volume [56; 57]. Meinzer et al. [58] adopted a simplified implementation of Kajiya's illumination model. Their approach is purely empirical, but they report high-quality images. Unfortunately, the excessive computational requirements of this method currently prevent its clinical application.

4.2.4. Stereo and perspective projection

A common enhancement for 3D display is the addition of stereoscopic viewing, which provides additional depth cues. Stereoscopy is customarily obtained by orthographic rendition of two images with a slightly different viewpoint. The angle between the two viewpoints determines the depth present in the stereo image. It has been proven difficult to estimate distances in depth; van der Voort et al. [53] report that most of the users report difficulties in using such stereoscopic pairs. This can be explained by the unusual combination of orthographic projection (assuming viewing at an infinite distance) and stereo (assuming nearby viewing) giving inconsistent depth cues. Visually realistic 3D images require perspective projection [59].

With perspective projection, the ray density decreases as one moves away from the observer. Consequently, the effective number of samples along rays decreases and care must be taken to avoid undersampling of the 3D data. Besides the undersampling problem, perspective projection does not facilitate commonly used methods for speeding up the ray casting step, which increases the required computational costs.

An apparent method for perspective projection is a preprocessing step that resamples the input data using a perspective transform. This allows for the subsequent ray casting step to use parallel projection techniques [30]. The approach is suitable for object order (forward mapping) algorithms and provides correctly sampled perspective images.

Westover [37] notes that perspective projection can be added to "splatting" methods without excessive costs; his renderer can deal with changing sampling rates of the input grid by appropriate resizing of the precalculated footprints.

The addition of perspective projection to backward mapping ray casting techniques has proven to be more difficult. An obvious approach is to choose high sample rates that provide correct sampling at the "yon" (far away) side of the volume. Unfortunately, this leads to a dramatic increase in the number of sampled rays as well as the need for an additional filtering step to correctly handle the supersampling at the "hither" side of the volume.

Levoy [60] proposes an algorithm that facilitates perspective projection by using 3D mip maps (a three-dimensional extension of the 2D mip map texture resampling method reported by Williams [61]). Precalculated colours and opacities are stored at different levels of resolution; during the image-order ray casting step, undersampling is avoided by decreasing the input resolution of the dataset (by choosing a mip map at a lower resolution level) where appropriate. Although objects at the yon side of the volume are rendered at a lower resolution, satisfying results were reported.

Finally, a simple approach for perspective projection for backward mapping methods was proposed in [62]. It uses a traditional ray casting step, but spawns additional rays when undersampling threatens to occur; results of adjacent spawned rays are then merged in back-to-front order. This approach seems very suitable for adaptive ray termination, although this has not been confirmed.

The majority of current implementations completely ignore perspective projection, since it is costly and does not provide dramatic improvements in depth perception when not combined with interactive stereo display. We expect that perspective will get much more attention when hardware becomes available that is capable of high speed visualizations.

4.2.5. Geometrically defined objects

Potential clinical applications of mixing geometric and volumetric data in a single visualization are easy to find, especially in the field of radiation therapy where superposition of radiation treatment beams over patient anatomy gives important information on the geometry of the treatment planning.

Since geometric and volumetric data have totally different spectral properties, great care should be taken when mixing them. There are essentially two approaches: combination of separately shaded surface and volume data and merging the two datasets into one dataset with subsequent ray casting [63].

The first method employs a hybrid tracer: Rays are simultaneously cast through the volumetric sample arrays and the polygonal dataset, after which the resulting colour samples are composited together in depth-sorted order. In this technique, a modified blending algorithm should be used since geometrically defined planes usually have infinite thinness (or at least are significantly thinner than the voxel size) [54]. Supersampling must also be performed to prevent aliasing along polygon edges.

The second method first converts geometrical objects into a 3D volume dataset by performing appropriate shading, filtering, and resampling. This transformation is called 3D scan-conversion [64; 54]. The filtering step guarantees a band-limited dataset, which avoids aliasing effects. The filtered dataset is then merged with the original volume, after which standard visualization techniques are applied on the merged dataset.

The best image quality is delivered by the hybrid ray tracer, since the filtering step associated with 3D scan conversion results in blurred polygonal edges. The hybrid approach is also suited for animation. With a constant viewpoint, we can separate the static volume data (*e.g.*, storing the calculated opacities and colours in a depth-buffer) from the dynamic geometrical primitives representing information as iso-dose contours or an "electronic scalpel". Combination of both data can then be done by Z-merging algorithms [63].

The addition of geometrically defined objects often leads to better insight in the 3D image structure, which suggest that geometric primitives may also be useful as diagnostic tool in the study of volumetric datasets.

5. Trends

Volume visualization techniques imply processing of an enormous amount of data, which prevents contemporary workstations from generating high-quality 3D images in a truly interactive manner. This partially explains the fact, that in spite of great interest from the medical community, routine use of volume visualization has not been established yet.

It is likely that interactive visualization of medical datasets will become reality in this decade. Special purpose architectures as Pixel-Planes 5 are currently providing near interactive speeds [65; 43]. Since the speed of computer hardware tends to double each year, adequate computing power should be available before the end of the century. As the size of 3D datasets poses a huge data transfer problem, only parallel processing will facilitate real-time interactive manipulation. Suitable algorithms for data caching and load balancing on parallel architectures have to be developed.

An important field is the design and implementation of good user interfaces for volume visualization. The window and mouse style of interaction, that has become popular in nearly all low- and high-end computer platforms, is also being adopted by manufacturers of medical imaging equipment. In the next years we expect a gradual change-over from primitive "keyboard and pushbutton" interfaces to graphical user interfaces (GUI's) that are adopted to the specific requirements of its medical users.

The standard window and mouse style interfaces are less suited for direct 3D interaction with volume data. Visualization techniques and hardware as well as suitable input and output devices should be developed that support such interaction. Examples of potential applications are interactive radiotherapy treatment planning, rehearsal of surgical procedures, and real-time volume imaging. The development of adequate 3D user interfaces and fast parallel graphics hardware are important topics in the "Virtual Reality" wave that has inundated the graphics community. In spite of the rapid advancements in this field, it will take several years before the techniques become both usable and affordable for clinical applications.

Shadowing and perspective are important cues to get an insight into the structure of complex 3D objects. Facilitated by the increasing speed of computer hardware, we expect the development of various algorithms that provide these possibilities and are tuned towards specific applications and/or special purpose computer architectures.

Problems related to registration and segmentation are not specific for the medical field; numerous research groups working on image processing are addressing related problems. Unfortunately, progress on automatic 3D object recognition and tissue classification is relatively slow and much work still has to be done.

Even if perfectly registered and segmented datasets are available, their 3D display would still pose substantial problems. There is a great need for good strategies that display the essential aspects of different datasets in a rendered image (data fusion). Some initial results that combine anatomical and functional information obtained from different modalities can be found in [66] and [67].

The progress in rendering techniques, in software development, and in computer hardware guarantees that volume visualization will stay a key development area in the 1990's. Perhaps the biggest challenge in this field is the development of clinically applicable strategies for integrated registration, segmentation, manipulation and visualization of multimodality datasets. Although this goal still seems far away, its achievement will be benificial to many areas of medicine.

Acknowledgements

This research was partially supported by the Dutch Ministry of Economic Affairs (SPIN grant (VS-3DM) 50249/89-01) and by the industrial companies Agfa-Gevaert, KEMA and Philips Medical Systems.

References

1. F. Zonneveld, S. Lobregt, J. van der Meulen, and J. Vaandrager, "Three-dimensional imaging in craniofacial surgery," *World J. Surg*, vol. 13, pp. 328–342, 1989.
2. E. Fishman, D. Ney, and D. Magid, *Three-Dimensional Imaging: Clinical Applications in Orthopedics*, pp. 425–440. Vol. 60 of Höhne *et al.* [68], 1990.
3. D. Ney, E. Fishman, and D. Magid, "Three-dimensional imaging of computed tomography: Techniques and applications," in VBC90 [69], pp. 498–506.
4. D. Ney and E. Fishman, "Editing tools for 3d medical imaging," *IEEE Computer Graphics and Applications*, vol. 11, pp. 63–71, November 1991.
5. S. Raya and J. Updupa, "Shape-based interpolation of multidimensional objects," *IEEE Transactions on Medical Imaging*, vol. 9, pp. 32–42, March 1990.
6. C. Liang, W. Lin, and C. Chen, "Intensity interpolation from serial cross-sections," in *SPIE Vol. 1092 Medical Imaging III: Image Processing*, pp. 60–66, 1989.
7. J. Parker, R. Kenyon, and D. Troxel, "Comparison of interpolating methods for image resampling," *IEEE Transactions on Medical Imaging*, vol. MI-2, pp. 31–39, March 1983.
8. M. Levoy, "Rendering of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 28–37, 1988.
9. G. Frieder, D. Gordon, and R. Reynolds, "Back to front display of voxel-based objects," *IEEE Computer Graphics and Applications*, vol. 5, pp. 52–59, January 1985.
10. D. Cohen, A. Kaufman, R. Bakalash, and S. Bergman, "Real time discrete shading," *The Visual Computer*, vol. 6, pp. 16–27, February 1990.
11. A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel, "A survey of architectures for volume rendering," *IEEE Engineering in Medicine and Biology*, vol. 9, pp. 18–23, December 1990.
12. K. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke, "3d visualization of tomographic volume data using the generalized voxel model," *The Visual Computer*, vol. 6, pp. 28–36, February 1990.
13. A. Pommert, U. Tiede, G. Wiebecke, and K. Höhne, "Surface shading in tomographic volume visualization: A comparative study," in VBC90 [69], pp. 19–26.
14. B. ter Haar Romeny, L. Florack, J. Koenderink, and M. Viergever, "Space space: Its natural operators and differential invariants," in Colchester and Hawkes [70], pp. 239–255.

15. B. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, 1975.
16. J. Coatrieux and C. Barillot, *A survey of 3D Display Techniques to Render Medical Data*, pp. 175–196. Vol. 60 of Höhne *et al.* [68], 1990.
17. A. Kaufman, *Volume rendering.* IEEE Computer Society Press, 1990.
18. R. Robb and D. Hanson, "Analyze: A software system for biomedical image analysis," in VBC90 [69], pp. 507–518.
19. H. Fuchs, Z. Kedem, and S. Uselton, "Optimal surface reconstruction from planar contours," *Communications of the ACM*, vol. 20, pp. 693–702, October 1977.
20. H. Christiansen and T. Sederberg, "Conversion of complex contour line definitions into polygonal element mosaics," *Computer Graphics*, vol. 12, pp. 187–192, July 1978. Proc. Siggraph.
21. Y. Shinagawa and T. Kunii, "Constructing a reeb graph automatically from cross sections," *IEEE Computer Graphics and Applications*, vol. 11, pp. 44–52, November 1991.
22. W. Lorensen and H. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *Computer Graphics*, vol. 21, pp. 163–169, July 1987. Proc. Siggraph.
23. A. Wallin, "Constructing isosurfaces from ct data," *IEEE Computer Graphics and Applications*, vol. 11, pp. 28–33, November 1991.
24. H. Cline, W. Lorensen, S. Ludke, C. Crawford, and B. Teeter, "Two algorithms for the three-dimensional reconstruction of tomograms," *Medical Physics*, vol. 15, pp. 320–327, May/June 1988.
25. W. Lorensen and H. Cline, "Volume modelling," in *Volume Visualization Algorithms and Architectures*, pp. 45–65, 1989.
26. H. Cline, W. Lorensen, S. Souza, R. Kikinis, G. Gerig, and T. Kennedy, "3d surface rendered mr images of the brain and its vasculature," *journal of Computer Assisted Tomography*, vol. 15, no. 2, pp. 344–351, 1991.
27. J. Udupa and G. Herman, "Volume rendering versus surface rendering," *Communications of the ACM*, vol. 32, pp. 1364–1366, 1989.
28. G. Frieder, G. Herman, C. Meyer, and J. Udupa, "Large software problems for small computers: An example from medical imaging," *IEEE Software*, vol. 2, pp. 37–47, September 1985.
29. K. Höhne, M. Bomans, A. Pommert, and U. Tiede, "Voxel-based volume visualization techniques," in *Volume Visualization Algorithms and Architectures*, pp. 66–83, 1989.
30. R. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *Computer Graphics*, vol. 22, no. 4, pp. 65–74, 1988. Proc. Siggraph.
31. T. Porter and T. Duff, "Compositing digital images," *Computer Graphics*, vol. 18, no. 3, pp. 253–259, 1984. Proc. Siggraph.
32. D. Ney, E. Fishman, D. Magid, and R. Drebin, "Volumetric rendering of computed tomography data: Principles and techniques," *IEEE Computer Graphics and Applications*, vol. 9, no. 2, pp. 24–32, 1990.
33. K. Höhne, M. Bomans, A. Pommert, M. Riemer, U. Tiede, and G. Wiebecke, *Rendering Tomographic Volume Data: Adequacy of Methods for Different Modalities and Organs*, pp. 197–216. Vol. 60 of Höhne *et al.* [68], 1990.
34. A. Levinthal and T. Porter, "Chap — a simd graphics processor," *Computer Graphics*, vol. 18, pp. 77–82, July 1984. Proc. Siggraph.
35. M. Levoy, "Volume rendering by adaptive refinement," *The Visual Computer*, vol. 6, pp. 2–7, February 1990.
36. M. Levoy, "Efficient ray tracing of volume data," *ACM Transactions on Graphics*, vol. 9, pp. 245–261, July 1990.
37. L. Westover, "Footprint evaluation for volume rendering," *Computer Graphics*, vol. 24, pp. 367–376, August 1990. Proc. Siggraph.
38. E. Fishman, R. Drebin, D. Magid, W. Scott, D. Ney, A. Brooker, L. Riley, A. Ville, E. Zerhouni, and S. Siegelman, "Volumetric rendering techniques: Applications for three-dimensional imaging of the hip," *Radiology*, vol. 163, no. 6, pp. 737–738, 1987.
39. J. Siebert, T. Rosenbaum, and J. Pernicone, "Automated segmentation and presentation

algorithms for 3d mr angiography." SMRM Poster 758, 1991.
40. F. Hottier and A. Billon, *3D Echography: Status and Perspective*, pp. 21–41. Vol. 60 of Höhne *et al.* [68], 1990.
41. J. Udupa and H. Hung, "Surface versus volume rendering: A comparative assessment," in VBC90 [69], pp. 83–91.
42. Kalender, Seibler, Klotz, and Vork, "Single-breathold spiral volumetric computed tomography by continuous patient translation and scanner rotation," in *Abstracts 75th annual meeting RSNA*, (Chicago), p. paper No. 1370, 1989.
43. R. Ohbuchi and H. Fuchs, "Incremental volume rendering algorithm for interactive 3d ultrasound imaging," in Colchester and Hawkes [70], pp. 486–450.
44. J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *Computer Graphics*, vol. 25, pp. 57–62, November 1990. San Diego Workshop on Volume Visualization.
45. D. Laur and P. Hanrahan, "Hierarchical splatting: A progressive refinement algorithm for volume rendering," *Computer Graphics*, vol. 25, no. 4, pp. 285–288, 1991. Proc. Siggraph.
46. J. Udupa and D. Odhner, "Fast visualization, manipulation, and analysis of binary volumetric objects," *IEEE Computer Graphics and Applications*, vol. 11, pp. 53–62, November 1991.
47. T. Foley, D. Lane, and G. Nielson, "Toward animating raytraced volume visualization," *The Journal of Visualization and Computer Animation*, vol. 1, pp. 2–8, February 1990.
48. B. Gudmundsson and M. Randén, "Incremental generation of projections of ct-volumes," in VBC90 [69], pp. 27–34.
49. A. Pommert, U. Tiede, G. Wiebecke, and K. Höhne, *Image Quality in Voxel-Based Surface Shading*, pp. 737–741. Computer Assisted Radiology, Berlin: Springer-Verlag, 1989.
50. A. Pommert, W. Höltje, N. Holzknecht, U. Tiede, and K. Höhne, "Accuracy of images and measurements in 3d bone imaging," in Lemke *et al.* [71], pp. 209–215.
51. M. Levoy, H. Fuchs, S. Pizer, J. Rosenman, E. Chaney, G. Sherouse, V. Interrante, and J. Kiel, "Volume rendering in radiation treatment planning," in VBC90 [69], pp. 4–10.
52. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics – Principles and Practice*. Addison-Wesley, second ed., 1990.
53. H. van der Voort, G. Brakenhoff, G. Janssen, J. Valkenburg, and N. Nanninga, "Confocal scanning fluorescence and reflection microscopy: Measurements of the 3-d image formation and application in biology," in *Proceedings SPIE vol. 808*, pp. 138–143, 1987.
54. M. Levoy, "A hybrid ray tracer for rendering polygon and volume data," *IEEE Computer Graphics and Applications*, vol. 10, pp. 33–40, March 1990.
55. M. Levoy, *Display of Surfaces from Volume Data*. PhD thesis, University of North Carolina, Chapel Hill, May 1989.
56. J. Kajiya, "The rendering equation," *Computer Graphics*, vol. 20, pp. 143–149, August 1986. Proc. Siggraph.
57. P. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing," *Computer Graphics*, vol. 24, pp. 145–154, August 1990. Proc. Siggraph.
58. H. Meinzer, K. Meetz, D. Scheppelmann, U. Engelmann, and H. Baur, "The heidelberg ray tracing model," *IEEE Computer Graphics and Applications*, vol. 11, pp. 34–43, November 1991.
59. M. Hagen, "How to make a visually realistic 3d display," *Computer Graphics*, vol. 25, pp. 76–81, April 1991.
60. M. Levoy, "Gaze-directed volume rendering," *Computer Graphics*, vol. 24, no. 2, pp. 217–223, 1990.
61. L. Williams, "Pyramidal parametrics," *Computer Graphics*, vol. 17, pp. 1-11, July 1984. Proc. Siggraph.
62. K. Novins, F. Sillion, and D. Greenberg, "An efficient method for volume rendering using perspective projection," *Computer Graphics*, vol. 24, no. 5, pp. 95–102, 1990.
63. A. Kaufman, R. Yagel, and D. Cohen, *Intermixing Surface and Volume Rendering*, pp. 217–228. Vol. 60 of Höhne *et al.* [68], 1990.
64. A. Kaufman, "Efficient algorithms for 3d scan-conversion of parametric curves, surfaces and volumes," *Computer Graphics*, vol. 21, pp. 171–179, July 1987. Proc. Siggraph.

65. H. Fuchs, M. Levoy, and S. Pizer, "Interactive visualization of 3D medical data," *IEEE Computer*, pp. 46–51, August 1989.
66. X. Hu, K. Tan, D. Levin, C. Pelizari, and G. Chen, *A Volume-Rendering Technique for Integrated Three-Dimensional Display of MR and PET Data*, pp. 379–398. Vol. 60 of Höhne *et al.* [68], 1990.
67. P. van den Elsen and M. Viergever, "Fusion of electromagnetic source data and tomographic image data," in Lemke *et al.* [71], pp. 240–246.
68. K. Höhne, H. Fuchs, and S. Pizer, eds., *3D Imaging In Medicine*, vol. 60 of *series F: Computer and System Sciences*. Springer-Verlag, 1990.
69. IEEE Computer Science Society, *First Conference on Visualization in Biomedical Computing*, (Atlanta), IEEE Computer Society Press, 1990.
70. A. Colchester and D. Hawkes, eds., *Information Processing in Medical Imaging*, Springer-Verlag, July 1991.
71. H. Lemke, M. Rhodes, C. Jaffe, and R. Felix, eds., *Computer Assisted Radiology*, Springer-Verlag, 1991.

SCIENTIFIC VISUALIZATION

J.J. VAN WIJK
Netherlands Energy Research Foundation ECN,
P.O. Box 1, 1755 ZG Petten, The Netherlands

Abstract. The aim of scientific visualization is to get insight in the results of simulations and measurements with interactive computer graphics. An overview of the different aspects of visualization is provided on the basis of the visualization-pipeline. The division of the visualization process in different steps, the interaction with the user, the use of hardware, and the different solutions for software are discussed. One of the greatest challenges in visualization is the visualization of three-dimensional flow. Several new techniques for this were developed at the ECN Visualization Centre.

1. Introduction

The rapid development of computer hardware and software has promoted the use of simulation as a tool for analysis and evaluation. The number of applications, the size as well as the complexity of simulations have grown exponentially in the last decades. Modern computer hardware has provided a solution for the generation of large amounts of data, but also given rise to a new major problem: the human interpretation of these very large data-sets.

An important way for researchers to get insight in the result of large scale simulations is via the use of computer generated images. The bandwidth of the human visual system matches the size of the data-sets. Further, computer graphics hardware has developed very rapidly: from plotters in the sixties, via line drawing displays in the seventies, to nowadays 3-D graphics workstations. With these workstations shaded images of complex models can be generated in fractions of seconds.

These developments have led to a new sub-discipline in computer graphics: *scientific visualization.* This term was first used in 1987 in an influential report of a committee of the U.S. National Science Foundation (McCormick et al., 1987). The report states that: "The goal of visualization is to leverage existing scientific methods by providing new scientific insight through visual methods." The keyword here is *insight*, insight of the researcher in his simulation. If the researcher has gained this insight, he also wants to communicate this to others. Presentation is therefore another, partly overlapping, goal of visualization.

In this article first a global overview is given of the current state of visualization. To this end the visualization process is broken down into a number of steps: the visualization pipeline. By means of this concept several aspects, such as user interaction, the use of hardware, and the solutions for software are discussed.

The ECN Visualization Centre is involved in the development of practical solutions for visualization problems, and in research to explore and shift the limits of visualization. One of most challenging application domains for visualization is Computational Fluid Dynamics. Several new techniques for the visualization of flow were developed at the ECN Visualization Centre.

2. Visualization pipeline

The visualization process is best described as a sequence of processing steps: the original data are sequentially processed in a pipeline, until images result (Upson et al., 1989). The typical steps in this process are:

- *data generation*: simulations or measurements produce data;
- *data enrichment and enhancement*: the data are selected, filtered, smoothed, interpolated, transformed, combined, etc.;
- *visualization mapping*: the data are mapped onto visual primitives (e.g. polygons, lines) and their optical properties (colour, reflectivity) are derived;
- *rendering*: the visual primitives are displayed on the screen after performing viewing transformations, hidden surface removal, shading calculations, and scan-conversion.

For the last step standard computer graphics techniques can be used. However, the process involves more. Other disciplines that play an important role are data base management, user interface design, and network technology. Further, the problems and conventions differ for each application domain. The input of experts from the application domain is therefore indispensable for a successful development of visualization solutions.

By means of this visualization pipeline we will consider a number of aspects of scientific visualization: the role of the user, the use of hardware, and solutions for software.

3. Interaction

The concept of a pipeline suggests that the display of scientific data is a one-off event. Obviously, in practice it is part of a cycle, as shown in fig. 1. The interpretation of an image will often induce that the researcher wants a different view on his data that provides more information. The user can control the visualization for each step of the pipeline. The most simple operation is to select a different view or a different position of a light source. Each aspect of the data can be visualized in different ways: a chemical bond can be shown as a line, a cylinder, or as a iso-potential surface; colour scales can be changed, etc..

The selection of the data to be visualized varies strongly with the phase of the analysis of the data. Usually one starts with a global overview, next interesting details are isolated and studied further. On the highest level the researcher can directly vary parameters that influence his simulation. This is known as *computational steering*, and cannot be realized yet for complex 3-D simulations. But we can expect that in the near future a designer changes the construction of

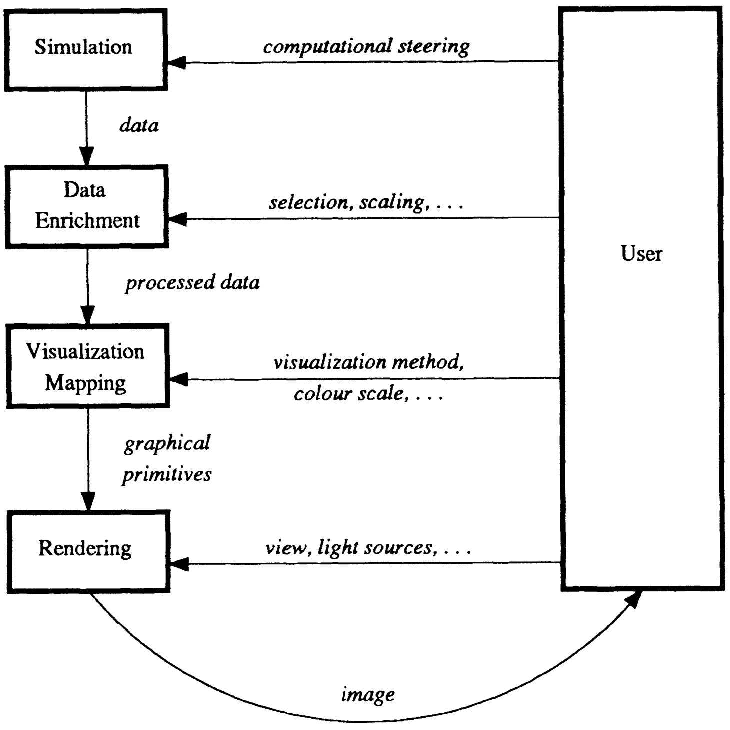


Fig. 1. User and visualization pipeline

his product interactively, while simultaneously the consequences for the occurring stresses are shown.

4. Hardware Allocation

A typical configuration for performing simulations is a combination of a supercomputer and a graphics workstation, connected by a network. It depends on the application how the visualization pipeline is mapped onto the hardware (fig. 2). If the simulation is not too demanding, the whole process, from calculation up to display, can be done on a workstation. A supercomputer can be used in one of the following ways:

- simulation: all results are sent to the workstation;
- simulation and data enrichment: processed data are transferred;
- simulation, data enrichment, and visualization mapping: graphics primitives are transferred;

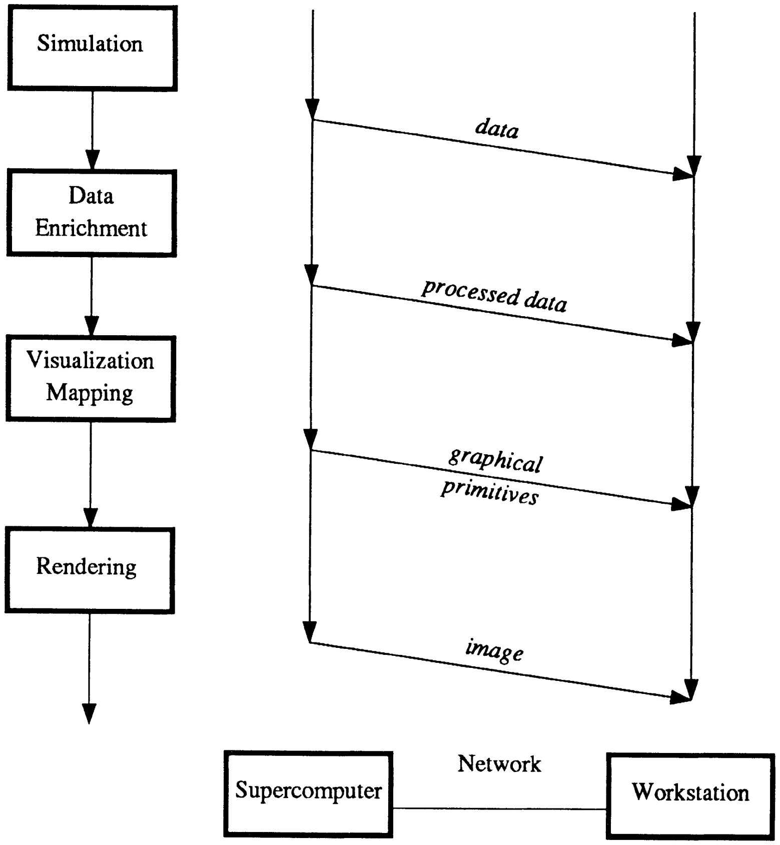


Fig. 2. Hardware and visualization pipeline

– simulation, data enrichment, visualization mapping, and rendering: images are transferred.

An advantage of the first option is that all data can be processed locally on the workstation. However, if the capacity of the workstation falls short, a better alternative is to pre-process the data near the source (option 2). The third option is often used: the X-Windows protocol is a flexible and portable solution for the transport of graphical information across networks. An important limitation is that the graphics primitives of X-Windows are two-dimensional, which inhibits the use of the 3D facilities of modern graphics workstations. PEX is a 3D-extension of X-Windows that allows a better use of the equipment.

If the demands on the image quality are very high, then the calculation of the images is a CPU-intensive process on its own. A natural solution is then to use the supercomputer for the full visualization process.

The amounts of data that are sent over the network vary strongly, dependent on the application and the allocation of tasks over the hardware. It is therefore possible that the network becomes the limiting factor for the cycle-time. This is another reason to consider the use of hardware carefully.

5. Software

A number of solutions is possible for the use of software for scientific visualization (fig. 3). On the lowest level are general graphics standards, ISO or de facto. Examples are GKS, PHIGS, and the Graphics Library (GL) of Silicon Graphics. The functional level of those standards is that of the last step of the visualization pipeline: the conversion of graphics primitives to images. If such a standard is used as a starting point, then custom programming is required for the remainder of the visualization pipeline. The advantage is that the visualization application can be optimized according to one's own wishes and insights. For example, the features of the hardware can be exploited, and the user interface can be tailored to specific wishes. The disadvantages are that the development of software is still costly, and the development of interactive graphics applications requires expertise. In many cases it is therefore advisable to use a ready-made solution.

The next step is the use of a library for scientific graphics, such as UNIRAS or DISSPLA. Such libraries offer a large number of options for drawing graphs, functions of two variables, etc., with many possibilities to adjust graphical features, such as the axes or the colour scales used.

The use of libraries requires an application program that calls the subroutines of the libraries. A different solution, rapidly gaining in popularity, is the use of end-user packages. Here the user enters a data-set, and can specify with menus, sliders, buttons etc. how these data must be visualized. Examples of such packages are Unigraph 2000 of UNIRAS, PV-Wave, and the Data Visualizer of Wavefront Technologies. The use of such packages is easy to learn, and gives quick results. A disadvantage is that the user is limited to the functionality that has been built in by the manufacturer. This will be no problem however in many situations.

In the ideal situation the researcher can use an environment for visualization that handles the complete pipeline, and which is both flexible and easy to use. The so-called *application builders* aim to provide a solution for this. Examples are AVS, apE and Explorer (Silicon Graphics). Here the visualization process is split into modules. Each module has a limited functionality and a small number of input and output channels. Typical modules are modules for reading data, calculation of iso-surfaces, the selection of colour scales, and for rendering. The (advanced) user can add modules, so that the functionality of such a system has no limits.

The modules are connected by visual programming. Each module is presented on the screen as a window, with buttons and other controls to adjust control parameters. The user can select with the mouse how the modules are connected.

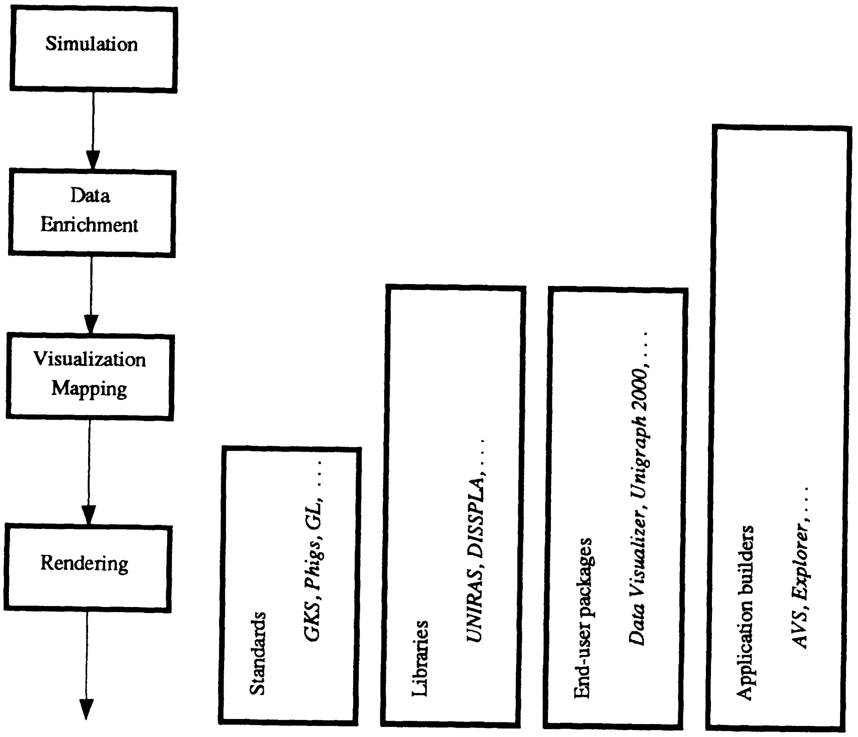


Fig. 3. Software and visualization pipeline

The resulting network of modules is a visualization application, hence the term application builders.

The concept of application builders is very promising: flexible, expandable, and relatively easy to use. They have appeared only recently, so the future has to tell if these promises will be fulfilled in practice.

6. ECN Visualization Centre

The preceding discussion shows that a single solution for visualization, optimal in all situations, does not exist. Probably it will never arrive, because the spectrum of applications is too wide: varying from a simple X-Y plot to the visualization of complex 3-D time-dependent fluid dynamics calculations. Therefore, several solutions are available at the Netherlands Energy Research Foundation ECN. For a large number of applications libraries and end-user packages for scientific graphics are sufficient. However, the number of applications that generate complex data sets, or with high demands on quality, speed, and ease of use, is growing fast.

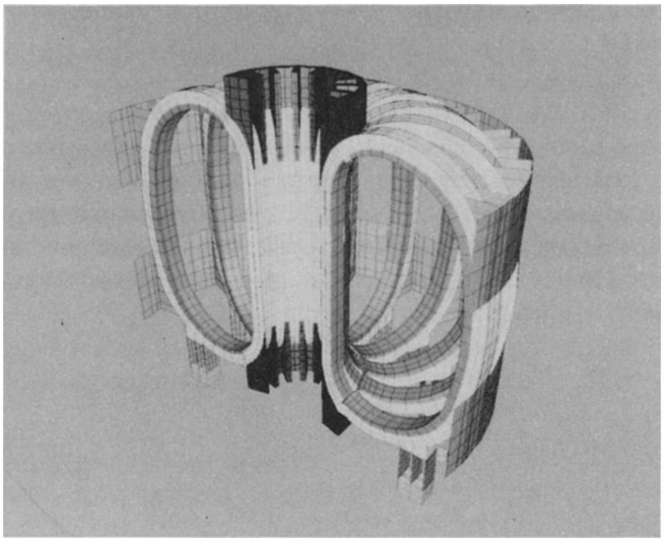


Fig. 4. Visualization finite elements

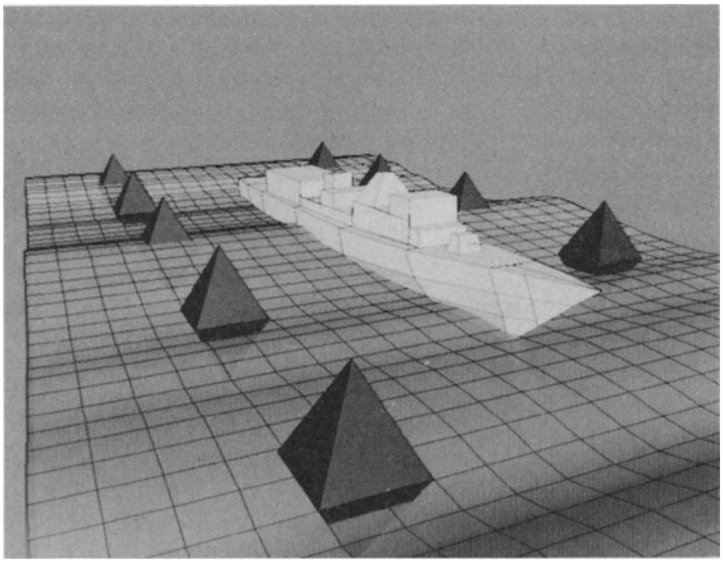


Fig. 5. Visualization ship dynamics

In 1991 the ECN Visualization Centre was established to supply an answer to this demand. In this context a high quality 3-D graphics workstation (Silicon Graphics VGX) and the Data Visualizer of Wavefront Technologies were purchased.

In some situations, it is worth while to develop a visualization package for one application. The reasons for this differ. Some users work very intensively with always the same type of data. A visualization package with an optimized user-interface and functionality leads then to a significant gain of time. In other situations the amounts of data are so large (initially or after the visualization mapping) that everything has to be done to use the hardware optimally. Another important reason to develop a package is simply that no ready-to-use applications are available on the market.

Fig. 4 and 5 show two examples of visualization packages developed at ECN. In fig. 4 nine coils of the Next European Torus fusion reactor are shown. The calculation of the stresses and distortions was done with ANSYS by C.T.J. Jong, ECN - Applied Mechanics. The image was made with ELVIS, a package for the visualization of the results of finite elements calculation. With ELVIS the stresses, temperatures, and distortions can be shown dynamically.

Fig. 5 shows an application for the visualization of the dynamics of ships, developed for MARIN (Maritime Research Institute Netherlands). The input of the package are for each ship spectral transfer functions for the six degrees of freedom. These are calculated by R. Dallinga and R. Huijsmans (MARIN). The waves of the sea surface are also described spectrally by wave models. The user can establish parameters such as the wind force and view direction interactively, so that he can get a good impression of the dynamical behaviour of the ship in the spatial domain.

7. Flow Visualization

The main topic for research of the ECN Visualization Centre is the visualization of flow. This is one of the greatest challenges in the area of visualizations. The main reasons for this are:

- the type of data: the results of a simulation are a combination of vector and scalar fields;
- the large amounts of data;
- the complexity of the structures that result from for instance the modelling of turbulence.

The research has led to several new techniques for the presentation of flow. Fig. 6 shows a visualization of a flow field with *surface-particles*. This concept was developed by J. Stolk as part of his Master's thesis project, with F.H. Post (Delft University of Technology) and the author as his supervisors (Stolk and Van Wijk, 1991). The use of particles is a well-known technique for flow visualization. The new aspect of surface-particles is that each particle is considered as a small facet.

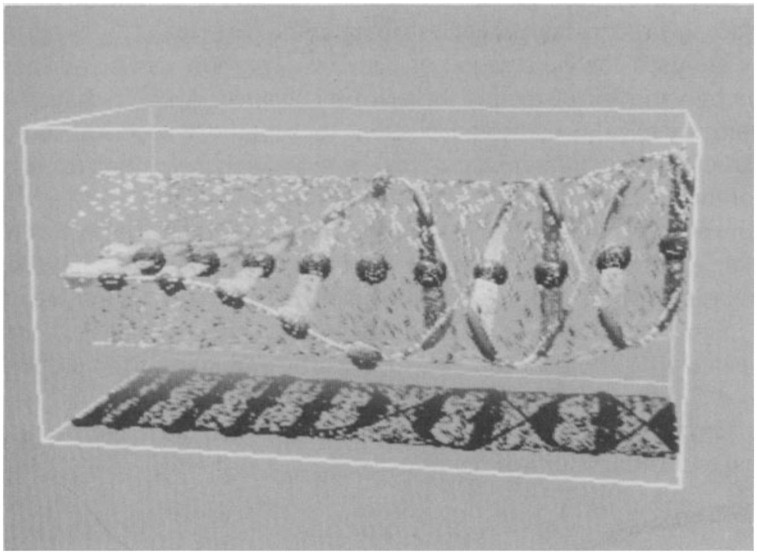


Fig. 6. Visualization flow with surface particles

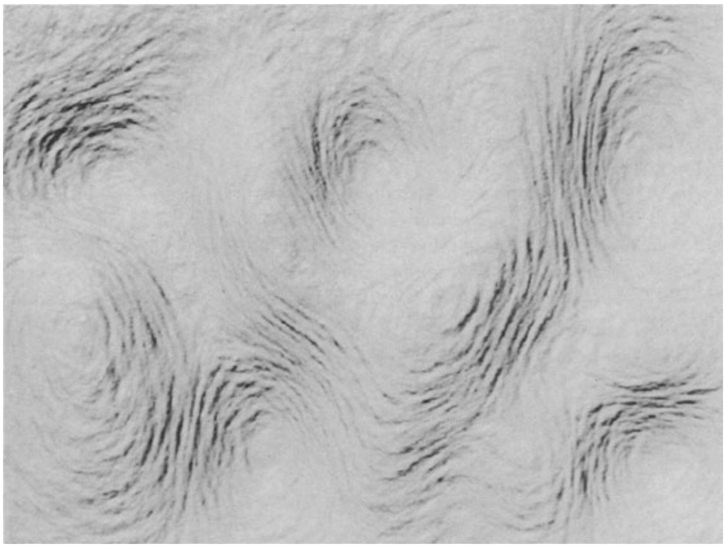


Fig. 7. Visualization flow with texture

It does not only have a position, but also a direction. This direction, or, the normal on the surface, is used to calculate the shading of the particle.

Sources are used for the creation of particles. The flow can be visualized in many ways by variation of the properties of this source. A circle-shaped source that continuously releases particles leads to a stream tube, a spherical source that releases particles at regular intervals gives a series of spheres, which distort under influence of the flow. Fig. 6 shows a number of possibilities.

Another result is shown in fig. 7. Here texture was used for the display of flow over a surface. This texture, *spot noise*, is generated by addition of a large number of randomly placed spots with a random intensity (Van Wijk, 1991). If the number of spots is large, then the individual spots can no longer be distinguished, and only texture is perceived. The data can be expressed by variation of the properties of the spot as a function of the data. For fig. 7 an ellipse was used, with the long axis in the direction of the velocity, and the ratio between the lengths of the axes proportional to the magnitude of the velocity.

8. Conclusion

Visualization is an important, if not indispensable tool for the analysis of simulations. An overview is given of the different aspects of visualization. Further, examples of applications and research are presented.

The current developments with respect to both visualization hardware and software are rapid. We can therefore expect that the role of visualization in research and development will become more and more important.

References

McCormick, B.W., T.A. DeFanti, and M.D. Brown (eds.): 1987, 'Visualization in Scientific Computing', *Computer Graphics* **21** (6),
Upson, C. et al.: 1989, 'The Application Visualization System: a Computational Environment for Scientific Visualization', *IEEE Computer Graphics and Applications* **21** (6), 30–42
Stolk, J. and J.J. van Wijk: 1991, 'Surface-Particles for 3D Flow Visualization', *Proceedings Second Eurographics Workshop on Visualization in Scientific Computing*, Delft, The Netherlands,
Wijk, J.J. van: 1991, 'Spot Noise – Texture Synthesis for Data Visualization', *Computer Graphics* **25** (4), 309–318

PICTURE ARCHIVING AND COMMUNICATION SYSTEMS : A SERVICE

R. MATTHEUS

Free University Hospital Brussels (V.U.B.)
Department of Radiology
Laarbeeklaan 101, 1090 Brussels
Belgium

Abstract. Medical imaging is an evolving discipline, after the discovery of Roentgen 100 years ago, the introduction of the computer totally changed the field. Medicine in general is much more specialized than before and computerization is a must. First automatization in the administrative sector of the clinic took place, next computers where introduced in medical equipment and now communication is playing a key role; this results in the need for dedicated services.

Images are generated, manipulated and communicated throughout the total hospital and even outside. The amount of information to deal with is 5000 times the magnitude of other clinical data. This results in the set-up of a complex heterogeneous environment as a service to provide images and other data on the locations where needed.

This chapter will give an idea of the evolution of the heath care model with the introduction and the needs of informatics. Communication will be described as the key component in this tele medicine revolution. A Picture Archiving and Communication System (PACS) service is described, given a lot of attention to the relations and impact of the Information Technology and Telecommunication (IT&T) industry on this complex system.

1. Introduction

We are speaking about Picture Archiving and Communication Systems (PACS) since 1982. First technical reasons, afterwards, integration and adaptation problems resulted in a difficult introduction of PACS.

In the beginning it was as a filmless radiology department, and now seems to result in a service for image management and communication on a departmental, hospital or even interhospital level. The health care model is complex and diverse. Medicine became more specialized and grew needs for communication of multi-media information. Technology movements will play a key role in the expansion and use of the application. Developments coming from the IT&T market as ATM, CD-I, will contribute to the next generations of PACS infrastructure.

These Image Management and Communication Systems infrastructures will be decentralized, heterogeneous, integrated with other departmental systems, open an phase upgradable.

P. Dewilde and J. Vandewalle (eds.), Computer Systems and Software Engineering, 397–426.
© 1992 *Kluwer Academic Publishers.*

2. Health Care System

2.1 Health Care Information Model

A health care information model focusing on imaging can be defined as consisting out of three levels as indicated in figure 1. Initiator level: requested phase, home preparation; the referring specialist ask for a dedicated examination and result, he asked for a service. The second level consists out of the service level, like radiology, pathology, lab... These have to perform a task and have to use resources like information: reports, old examinations, new examinations and equipment.

Until now a lot of R&D was done regarding the communication between and in these two lowest levels. The initials want to use a lot of services together, to match results for better diagnosis and treatment.

The communication problem is becoming complex and involves the need for multi-media, but until this is solved PACS will only have a limited value.

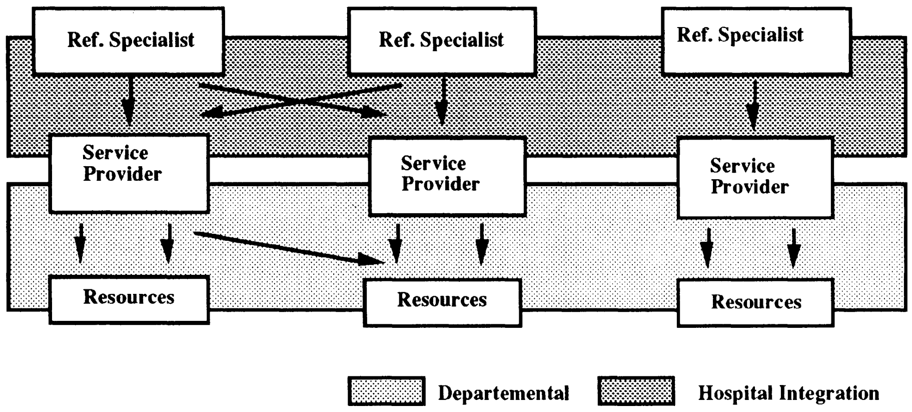


Fig.1. Information flow in the health care system. Referring specialist asks for a specific service, which will be preformed by the service provider using resources. Information coming from different service providers can be needed to preform a diagnosis, not necessary in the same hospital. Until know informatization on the departmental level is taking place, but hospital integration, by means of communication should be stressed.

2.2 Health care information flow

Medicine, and specifically Primary Health Care, have to respond to the changing patterns of population, disease and new treatments, to the demands of medical ethics and the law, and the economic pressures and expectations.

It is essential that a new application of technology either improves the quality of health care or increases the efficiency of providing care, preferably both. If it is to achieve successful implementation, it is necessary that it is socially acceptable and economically justifiable. Without these characteristics it will not be marketable and will not improve the competitivity of the producing industry. Subsequent exploitation of the investment requires the assurance of a long-term market, and increasingly an international market, to achieve efficient levels of production and marketing. While it is generally accepted that information and telecommunications technologies offer new possibilities in the provision, management and planning of medicine and health care, developments in information have almost inevitably highlighted the importance of the protection of the individual in terms of privacy and confidentiality. Privacy, in a health information context, means the right of the patient to decide who has access to information about him and the protection against the misuse or unjustified publication of that information.

In seeking solutions to the challenges which face us, we can make use of information and communication technology in a very productive way, but will depend on developing systems and the necessary software to support them, as well as taking advantage of the rapid developments in hardware. There has been an increase in the need for communication between hospitals and general practitioners as the pattern of health care has changed with the reduction in the number of hospital beds available and reduction in length of stay in hospital, but increase in the rate of admission to hospital.

Medical Informatics has been generated by two rapidly evolving disciplines, medicine and informatics, and by the need for communication.

Since the introduction of micro-electronics, medicine has known an unprecedented evolution. Informatics is made up of different layers, i.e. software and hardware, which do not evolve in a co-ordinated manner. The development of PACS has been significant in the past decade. Until sometime ago, the aim of PACS had been to manage medical images efficiently. But recently, as the technology evolved, there have been desires for adding new dimensions to the system, as for example coupling it with the Hospital Information System (HIS) and/or Radiology Information System (RIS), which is what some authors call "Hybrid PACS".

2.3 Complexity of the health care system.

A health care system, like any form of social organization, reflects the historical, cultural, and political as well as economic context within which its develops.

- the character and modes of the population
- the historical process of evolution
- the past and present government structure
- the central components of the national economy
- the source of financing
- the structure of medical education
- the strength of personnel unions and professional associations

Due to this, medical informatics is quite complex and applications need to be carefully mapped to a certain health care system. Terminology and scenarios understanding and definition are crucial to compare, integrate and harmonize the informatization of a health care system.

2.4 Scenarios

Considering the vastness and complexity of the health care environment, the medical information exchanges can only be defined in an abstract way.

One of the most important difficulties in medical informatics is the abstraction of the health-care environment. If we want to develop a generic model for the radiology department, with as target to build a PACS service, the first should be requirement specifications. The problems arise already at this first step. Terminology and other misconceptions between the to disciplines: medicine and engineering, result in quite unuseful specifications. One of the major problems is the level of abstraction and a clear understanding of each item in the requirement process. As a useful way to avoid this I would suggest the scenario description. Based on it, a common level of abstraction is defined and more basic elements can be described in the same way. An medical examination in Belgium, for example differs from an examination in France . The way medicine works is quite different, e.g. how is it organized, how is the equipment paid, etc... The only common goal is to make the patient better. Scenarios will help modelling, self education and bring understanding of the components for mutual work on the on the requirements, and their priorities.

A scenarios can be very general like sent image examination for diagnosis to other specialist; tele-radiology or quite detail sent selected image and related data by means of public networking to a specialist his home workstation.

3. Complexity of information

3.1 Medical information objects

Medical practice and patient care are based on information generation, collection, and exchange among the care providers. In the past two decades, we have witnessed tremendous growth in the number of medical specialities, many of which became additional sources of information. While this specialization improved the quality of treatment, it also created a communications nightmare. Information that is not communicated to the users in a timely and efficient manner cannot have any value no matter how significant the finding might be. Efforts and investments that went to developing the information is wasted.

So far, the medical community has placed more emphasis on generating useful information for patient care than on sharing or managing it. Centralized management of information has been the norm as it was in other industries in the past. As in other fields this changing flow of information will cause organizational transformations and power shifts. Providers of health care themselves must develop transformation strategies using information management technologies both to enhance the quality of care and to more efficiently use of health care resources .

Radiology services pose one of the most difficult problems of information management, because they generate large amounts of data and they must be managed rapidly. Radiology is one of the most widely used diagnostic services, with 50-70% of the patients entering a hospital receiving some form of radiological examination. Radiological information presents a more acute problem because it must be managed in a timely manner. On a single day a radiology department like that of the University Hospital of Brussels (VUB) may handle more than 1.500 sheets of film, including both old and new images. The management and distribution of radiology information and radiological images are integral parts of patient care. Though the new imaging technologies have given the radiologist a powerful set of new diagnostic tools, however the quality of radiology service has not experienced similar revolutionary improvements. In fact, the use of many imaging modalities has imposed additional difficulties to the management, which is already overburdened by the massive amount of film and supporting data to be archived and distributed. When time passes, also other departments will generate images and make this management issue more critical.

3.2 Images

In general, imaging systems deals with two main data-types, image data and image related data, both of which are normally related to physical phenomena The image data can be: grey-scal images, colour images, image sequences... Image related data can be look-up-tables, text, graphics, regions of interest, patient and examination related data can also be continuous physical variables, slice thickness...

A digital image is one that has been converted into numerical values for transmission or processing. A matrix is a square series of boxes that gives form to the image. The individual matrix boxes are known as picture elements or pixels. Pixel size determines resolution. In medical imaging each pixel value corresponds to a three-dimensional volume of tissue, known as a voxel. A elementary image data type can be define as falling in to the classes:
Raw - arrays of pixels, with pixels being of a basic type like a bit, byte, depending on the gray or colour levels. Images can have different dimensions depending on the application.

3.3 Medical Images

Medical Images, like each digital image, consist out of pixels. The spatial resolution is the number of pixels, typically 256*256, 512*512, 1024*1024, 2096*2096. The contrast resolution, i.e. the grey scale, of one pixel is typically 12bits (4096 grey levels).

In medical images we deal with a lot of information which is not comparable to normal administrative data. In particular we are not interested in one image but in a complex set of images, one study can be 5,40 or even 200 images, it can also be a dynamic study or 3D set.
The minimum amount, for 1 examination, depending on type and application, between 10 and 30 Mbyte. This data should be seen as one unit for manipulated, storage, display and transmission of these sets.

At this moment there is also no abstract description of the context of an image object. Actually study is being done, in the direction of image description by content. If we compare this with an abstract of a patient we can represent this by his patient ID.

96% of the information used in the hospital, is non image data but the amount of information is just opposite. 96% will be image/bytes and only 4% other patient data. This to illustrate that for medical imaging a mostly other technical world need to deal with these problems. In the medical world an images consist not only out of bytes representing the pixel data, but also related data like, patient name, specific acquisition data belongs to the image data or format.

3.4 Multi-media

When one considers the procedures taken in caring a patient, it is obvious that many forms of information are associated. The doctor will first talk to the patient taking note of his complaints, his medical history and background. In most cases the patient's blood and urine will be examined, images taken where appropriate and other more specific examinations performed as the encounter progresses. The information produced during this encounter includes sound, vision, image, text, smell, and sense of touch. Presently, these informations are mostly recorded in a written form according to the physician's impressions. The physician will try to be as accurate as possible but a third person will never know what he exactly observed at that time. Even the physician himself will not keep track of all his cases in his demanding career. If these informations could be kept in a way that is more objective, permanent and easy to retrieve it would be of value. This different types of data are multi-media data. Table I give an overview of the definitions and relations of these data elements.

TABLE I.
Components and relations in the multi-media domain.

Horizontal components	one or more representation media
Vertical elements	characterized by implicit or explicit links among the representation elements
Multi-media	multiple representation media. In enumerating media, the distinctions are based on the nature of the information.
Hypermedia	focus on the explicit links between representation elements.

Multi-media and hypermedia must be considered orthogonal characteristics. Clinical data can be in a variety of formats, in the form of voice, of image, of tracing text. This creates the need for multi-media standards, and sophisticated terminals, with multi-media and processing capabilities.

Information (component of Multi-media) broadcast: Different signals are tightly coupled and synchronized so that they can effectively be considered as a single medium.

Information (component of Multi-media) production. The different components can be processed and edited separately

Multi-media will play an important role in the near future. The total medical information process is based on documents, voice, images, text and video all multi-media components. The compact Disk (CD-ROM) and CD-Interactive (CD-I) will be a base for further expansion of the important market.

4. Implicit Role of Communications in health

4.1 Communication the kernel

In health care, recent decades have seen a considerable acceleration driven by scientific and technological progress as well as a strong and lasting commitment of large resources to this objective. Some of the new means already available and the potential of new developments in the coming decades may well revolutionize health care. This obviously has social consequence but it is also of considerable political and economic importance. Europe can pride itself on a leading role in some areas of health care and the related scientific domains. However, as the need increases for concentration of efforts, skills, facilities and financial resources in R&D in related domains such as biotechnology, medical informatics and telecommunication in medicine, Europe is rapidly falling behind. Cooperation between scientists, research centres and increasingly with industries and telecommunication authorities is becoming a necessity to engage and stay in this high-technology domain.

The health care sector relies on the collection, communication and management of large quantities of data, which can be in any format, from numerical data to voice, images to video. Such data is mostly confidential, and its quality is essential for decisions that involve life and death.

The demographic evolution, the change in the tastes and needs of patients, and the development of health services have led to the creation of a wide area network of health services. Patients and professionals interface this network at multiple locations and need real-time support for their information and decision needs. In emergency care and in home care, for example, access to the network has to be done in mobile conditions.

Health care also means working in remote regions and in the least developed regions. Professionals who are willing to work in such conditions often lack access to appropriate expertise to treat their patients and have great difficulties in keeping abreast with continuing medical education and research.

Emerging information techniques and telecommunication feasibilities, like Integrated Broadband Communications (IBC), wide area information services, mobile communications, satellite communications, communication standards and protocols, as well as new basic services and value added services (packet switched data networks, circuit switched data networks, teletext, electronic mail, videotext and other standardized telemedicine-oriented services) offer new options for improving communications in the health care sector, for creating an integrated health environment and for creating a market for new types of health services.

The health care sector relies heavily on the collection, communication and management of large quantities of data, originated from patients and from the operation of services. Much of the data is contained in images, which create great storage and processing problems.

Data and information have to be shared by a wide network of users and organizations. Clinical and administrative data is produced in many locations: hospitals, health centres , general practice surgeries, nursing homes, community organizations, the home of the patient . Present trends towards primary health care, home care and self help will make this characteristic even more salient. With the single market of 1993, it can be expected that mobility of patients increases and that international communication of medical information becomes a major need. A wide variety of professionals, with distinct background and practices, is involved in care and has to access and update clinical and administrative data quickly. In case of an emergency, real time communication and data processing is often required. Moreover, because patients and health care professionals have such a variety of cultural background, interactive interfaces are required with any new telecommunication system.

4.2 Telemedicine.

Telemedicine is a hybrid word adding the prefix "tele", meaning " a distance", to medicine. This conveys the basic meaning of the terms which is, " the support of medical expertise by communications technology so that patients or doctors can readily access the best possible advice".

Telemedicine means organizing and integrating information technology in such a way that resources outside the local organization can be used systematically in the activities of health service. What is needed to achieve this, includes the development of systems which can be used as " glue" to link different existing data systems.

The development towards a European medical record architecture, the development of efficient navigation strategies, the development of intelligent systems which will determine what the user really wants, and will allow the user to sift through the distributed data to find relevant information.

The health care system is composed of a patchwork of departmental services with relatively weak integration, which have developed at different paces. Much effort has to been put into creating minimum standards to enable efficient communications within the system, and with other related systems as, for example, the health insurance sector.

Telemedicine is the integration of two well-established disciplines, the medicine and the tele-communication. For the use of the European people the integration has to take place within a modified environment, the European Telemedicine Infrastructure.

4.3 System approach

A computer system can be viewed, at least abstractly, as a number of layers: a command language, applications, application services, and hardware.
A similar view can be adopted for open systems. As with an operating system, the topmost level represents the shell or user interface. In the context of an open system, such an interface may be dependent upon the individual user. Beneath the interface layer lies the collection of applications available to the user. The applications, in turn, make use of a number of application services. These services provide mechanisms and means for ensuring and aiding the operation of applications within an Open System and across a number of different computer systems. Three classes of services are particularly important: database management services, communication services, and directories.

Database management services and associated databases provide the basis for the distribution and management of data over multiple sites. These services also provide the means for user and/or application query of databases, local and remote, reliable transaction services used by applications, back-up and recovery.

Communication services provide the means for routing, file transfer, remote process execution, etc. It also provides the basic mechanisms for message transfer between applications at different sites.

Finally, directories provide the means for identifying the entities within the system. In particular, they allow one to identify users and to name system-wide objects, specify their properties, location, associated privileges, ownership, etc. They can also be used to aid in the development, maintenance and operation of the entire system. Together, this primary set of services provides the basis for additional application services, such as network resource management or performance analysis.

It is known that medical information systems are very expensive to develop and to implement, in terms of design and maintenance. One can also state that the applications keep no pace with the technology and so standardization of developing and modeling healthcare systems is extremely important. It is a must for interchangeability, for transportability and to keep costs down.

It is important because in Europe there are two trends. First of all the ground systems will decline and the vendor systems will incline over the next years. Hence the need for a framework which fits everything together. Building a health care information system is very complex, and one has to understand that to build one unique composition at an European level is not obtainable, partly due to the different policies and also due to different strategies and scenarios used. Integration is primordial and one has to follow a top-down approach, where one is starting from the healthcare environment and goes down to overwhelming detail and describes all operational tasks. By selecting and subdividing one defines a healthcare information system architecture. Further on, one defines common conceptual schemas and at the last level there is the physical design, which is not the aim of this task.

The other trend is that healthcare information system framework reflects the different bodies and organisations within Europe: it can cover laboratory or healthcare medical institutions. The common concept of the schemas is the logical model with final data descriptions and processes. What is important is that, defining from the HCI-framework down to the common conceptual schema, the degree of complexity increases, but that the higher one goes, the higher becomes the level of abstraction.

5. PACS - IMACS: A definition

5.1 PACS

The term Picture Archiving and Communication Systems became somewhat misleading for many current developments, as emphasis is given to archiving which is certainly not any more the main reason for such an environment. Image Management and Communication Systems is a much more accurate description. It is the PACS concept, but looked at as a service.

5.2 IMACS

Image Management and Communication Systems: describes an advanced image based medical information system. The system can manage varieties of medical images and is capable of retrieving and displaying alphanumeric information from the conventional medical information system, and maybe other Multi-media data. The IMAC system could be one of many configurations ranging from a single module capable of supporting a small medical operation, to a network of modules shared among major medical centres. IMACS covers the total hospital and PACS is more radiology restricted.

6. Levels of Application

6.1 Generic configuration

The generic configuration applies to all organizational levels, but the details rise in complexity, where hardware and communications are seldom the problem. The difficulty arises with image and related data linkage and relating to an increasing amount of diverse data and records, and when controlling the processing of similar or related images and related data. Table II gives the different classes, there needs and problems.

Functional Level or sub-departmental: At this level, the image system has a singular, well-defined scope and purpose. A functional level can be defined as a section where the same type of examinations are done, which indirectly results in the same type of examination equipment. Local reporting, manipulating, discussing and processing takes place.

Departmental Level: At this level, the scope is still singular, but the operations are more diversified. More archived information will be consulted, comparison between old and new images takes place. A higher level of integration of the information occurs.

Enterprise Level: At this level, all systems in a hospital or in its organizational equivalent are integrated: for example, communication and other services are available in a medical domain, diagnosis can be done in a university hospital, the treatment at a local hospital.

TABLE II.
Organizational levels of use. Image systems work the same at all organizational levels, much more so than conventional handling, but the technical differences are critical and must be built into lower-level systems to support the integrations that are intended for higher levels.

	Scope of activity	Requirements and problems
Functional level	* Singular, well-defined purpose; often unrelated to other functions * Usually in support of dedicated needs	* From 1 to 4 workstations, *Awareness for integration
Department level	* Singular activity that is broader in scope * Often marked by an intense number of acquisitions and accesses * need for more information integration	* One or more networks of workstations * Hundreds of details are often overlooked during planning
Enterprise level	* Covers every aspect of an organization that heavily relies on images * Can be narrowly or widely focused	* Can be any size, but more often than not a series of departmental-level set-ups * Integration of components can be difficult * Should be limited to dedicated applications

6.2 Why medical imaging ?

In order to change the film based systems, PACS must bring major advantages in clinical use and be integrated in the whole chain of medical events - inside as well as outside the department of radiology.

The clinical advantages of PACS are :

- *Better (easier, faster) access to patient information, including archived images.* Using the conventional system, primary reporting is efficiently performed, but the access to previous examinations, that must be fetched from archives, is far more difficult and hazardous. It always introduces an important delay in the diagnostic process. Comparison with previous examination is a key issue in medicine in several situations : in oncology, where it represents the basic tool for evaluating the efficacy of the treatment; in orthopaedics; for medico-legal evaluations; in intensive care units.

Moreover the time between : when the patient consults or is hospitalized, and when an efficient treatment is instituted, is an important factor. Shortening that interval is at least positive for the psychological comfort of the patient; in some situations, it can be medically important or even life-saving. It is in addition a factor of cost saving for the health care system, especially for in-patients (by shortening the hospitalization).

Therefore, everything that can shorten the delays in the diagnostic chain, represents a substantial qualitative advance, as well as a cost saving for the health care system. The PACS technology can here deliver a major clinical improvement.

- *Image manipulation capabilities.* Keeping the image information from the digital modalities in digital form offers the possibility to use the images as input for different types of specialized diagnostic exploitations (e.g. multiplanar reconstruction from magnetic resonance, 3D data set, 3D reconstruction, automated image analysis) or for computed procedures for therapeutic aid (radiotherapeutic treatment planning, stereotaxy, surgical simulation, ...).

- *Diagnostic decision support - scientific and educational purposes.* Digital archiving constitutes, especially if a dedicated indexing technique is used, an ideal basis for scientific and educational retrieval and collection. PACS technology also lends itself very well to a coupling with decision support systems.

7. Factors and Trend Lines

Cost Reduction: The price of just about every element associated with PACS-IMACS is declining. This element has been part of the history of data processing since its inception, and there is no reason to believe it will stop. Since only very recently, the system developers based most of the developments on IT&T products. It results in cost reduction and more migration paths in the future.

Wide-area network extreme high-density trunk lines: Eventually, high-density communications trunk lines will be able to carry traffic between computers and other stations -as if they were integral units of a single computer. This is especially important for PACS because of the long record lengths and the resulting pressure to distribute storage in systems where central storage could be more efficient and contribute to improved interior lines. Also in this area IT&T should play an important role.

Workstations: Workstations are available with resolutions acceptable for a lot of applications, the addressing possibilities are becoming acceptable. Network integration becomes more easy. The user-interface had a lot of attention and even standard tools become available.

The factors have arisen from within the computer industry. Some of these factors have always been present, and others are strongly related to the advent of practical image processing:

A renaissance for the computer industry: It is only recently that the medical imaging industry and system integrators begun to invest in this market segment.

The pressure to regain international economic leverage: The EC is investing a lot in this and related fields in the 92 internal market Frame work. Pilot project like EurIPACS will play a crucial role during the next years in setting up a PACS service.

The sinergy with related industries: A partial relation and sinergy is coming from the IT&T market and electronic document industry. Communication and Multi-media will play a key role in the near future. The advances in the public networking (ISDN/B-ISDN) and HDTV will stimulate the tele-medicine applications.

Strategic push and pull: The corporations and agencies that are first with image processing are pushing technology, and the balance of the competition will be pulled into the maelstrom just to keep pace. Perhaps it has always been this way, but the image processing momentum accelerates the process.

Open architectural environment: Most vendors have maintained an open architectural marketing approach, insofar as the realities of hardware and software permit. This stance has encourages new partnerships among them. In turn, this seems to accelerate technological research even more strongly than the strategic push and pull factor.

8. The major PACS components

8.1.Medical Images: There life-cycle.

Different stages can be distinguished, starting with image creation going to image communication, manipulation, processing to image display and storages. The components of the image-life-cycle are explained in table III.

The data flow in an imaging environment is reflected in the figure 2, based on the working draft, information technology - computer graphics - image processing and interchange (IPI) ISO/IEC. In the area of medical imaging special emphasis should be given to the information generation environment and image-to-image data conversion. Data integrity is an important aspect, where destructive and non destructive image manipulations are strongly dependent on the application.

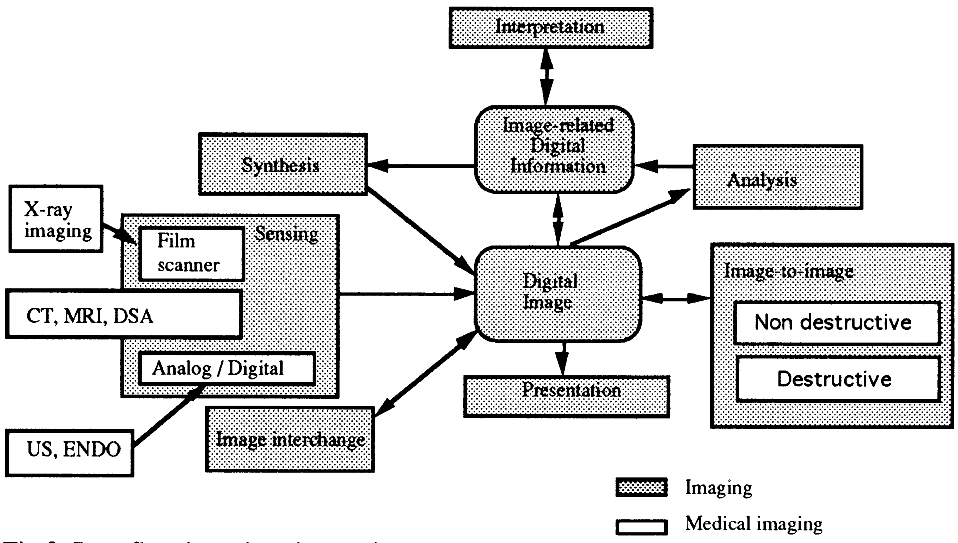


Fig 2. Data flow in an imaging environment.

TABLE III.

Image sensing	transformation of real-world image information to digital image; e.g Image acquisition device, scanner,...
Image presentation	transformation of digital images to real-world image information; video monitors, printers...
Image-to-image	transformation of digital images to digital images; e.g. grey value enhancement, edge detection,...
Image interchange	interchange of digital images among imaging systems; this serves for the communication of digital images and related non-image data among imaging systems.
Image analysis	transformation of digital images to non-image data; this encompasses basic functions like histogram generation, mean value...
Image synthesis	transformation of non-image data to digital images; this encompasses functions such as the rendering of lines, creation of test images...

8.2 Image Acquisition

Radiography is an art form that has been used to investigate wide array of persons, animals and objects. Understanding visual perception is an important aspect of clinical radiography. This understanding helps to overcome misperception of an image. The human eye is designed to gather light, focus it, convert it to nerve impulses and transmit it to the brain for processing.

Threshold detection is a visual phenomenon involving the perception of extremely small or faint details. The boundary effect occurs because the visual system has difficulty perceiving contrast differences that are distant from one another. When densities are adjacent, small differences can be perceived. When thee eye perceives a boundary, a change occurs in the intensity of the impulses sent to the brain. This change in intensity creates an effect known as edge enhancement, which makes the boundary appear distinct. Contrast perception is dramatically increased by eye motion or scanning the image. Pattern recognition involves perceiving combinations of details that can be defined and classified towards a diagnosis. It helps the radiologist compare mental images of patterns with medical knowledge.

Conventional radiography. (XR). Attenuation is the reduction in the total number of x-ray photons remaining in the beam after passing through a given thickness. The composition of the human body determines its radiographic appearance. When studying the absorption characteristics of the body, four major substances account for most of the variations in x-ray absorption: air, fat, muscle and bone. Photographic materials are photo-sensitive, or capable of responding to exposure by photons.

New developments where made possible with the introduction of informatics. The method by which these imaging modalities have been computerized is digital processing of imaging information. A detector of some type must be used to acquire the image information.

The primary mathematical method used in the creation of computerized medical images is the Fourier transformation. Convolution is the process of modifying pixel values by a mathematical formula. Deconvolution is the process of returning the pixel values to their original level by a reverse process.

The quality of the data acquired from the image receptor is measured by its frequency, contrast and noise. The density and contrast of the digital image are controlled by varying the numerical values of each pixel. The window level controls image density. Window width changes the gray scale expansion or compression and therefore controls contrast. Resolution is controlled by the matrix size.

Computer tomography (CT) produces a digital tomographic image from diagnostic x-rays. The basic principle of CT involves digitizing an image received from a slit scan projection of the patient's body and then projecting it back through mathematical algorithms. The combination of the transverse sectioning procedure with slit scanning produces an image of significantly better quality than that available with other imaging methods. Primarily the image differentiates various types of soft tissues.

Digital subtraction angiography (DSA) combines the digitization of an image with subtraction techniques.

Magnetic resonance imaging (MRI) uses magnetism and radio frequencies (RF) to create diagnostic sectional images of the body.

Ultra-sound (US). The application of ultrasound in medicine is based on the sonar principle. Acoustic waves are transmitted and reflected at interfaces according to the change in acoustic impedance.

Positron Emission Tomography (PET) scanner maps the distribution of pharmaceuticals labelled with positron emitting isotopes in order to construct detailed images of organ metabolism, physiology and function.

The only way to convert the conventional x-ray film to a digital one is by means of a laser digitizer. This laser scanner can in 6 seconds convert a film to a digital image of 2kx2k 16 bit, which is not too bad for most procedures. Major problems occur in operability, which is labour intensive; in throughput limitation, due to the time it takes to download the data to other mediums, which result in one examination of 6 images taking about 4 minutes; and in the way patient data is entered to the image.
The phosphor plates (DR) are a solution to some of these problems.

Digital Radiography (DR) or photo-stimulable radiography uses photo-stimulable image receptors made of barium fluorohalide screens that can store a latent image which can be released upon stimulation by light. They can be exposed in cassettes with diagnostic radiographic equipment.

All of these systems have a computer involved in the image generating process. This control and image processing computer is in most of the existing systems, dedicated for this tasks. Only in very recent development versions, networking possibilities are possible, even then without a clear interface, or dedicated software. The only advantage is that even with very tricky solutions this is the only way to have a digital link.

The other group of systems like Ultra Sound have an embedded dedicated computer involved without communication possibilities. The only way then is to convert to digital by means of an A-D convertor or frame grabber, which can results in information loss.

Examination type: Images are central objects in a medical environment like discussed in section 3.3. A clinician will look at a patients examination consisting of several images. This group of images can be characterized by 8 parameters. Based upon this 8-tuple, minimum requirements can be calculated for image-computers (resolution, manipulation functions, storage). Compression is not discussed hear, because compression can result in information loss, which is clearly a legal problem in medical imaging. A total reversible compression of 4:1 will be possible and is sometimes used. Table IV gives an example of different examination classes acquired at the VUB hospital in Brussels in 1991.

Definition of a 8-tuple (M,E,N,S,R,D,T,C) : Examination_type

M : {CT,MR,US,DR,XR,DSA} modality source
E : {body, head, neuro, cardiac, abdominal, vasc, bone} examination
N : maximum number of images
S : maximum number of marked images
R : {64,128,256,512,1024,2048,4096} matrix n x n
D : {2,3,4} display technique (depends on acquisition)
  2 : 2D image (x,y)
  3 : image of a 3D set(x,y,z)
  4 : image in a dynamic 3D set (x,y,z,t)
T : average examination time
C : {8,12} contrast resolution in grey level (bits per pixel)

Statistics: Different data types can be defined in a PACS network: (1) image data (2) image-related alphanumeric data, mostly in the image header, (3) patient related data and (4) control commands. The image data can be grouped in classes. Due to the different nature of the types of data and to the different requirements concerning the management and the flow, the network can be partitioned into message, image data and medical related data.

G : generated image data for one examination (Mbyte)
F : Frequency of the examination (Examination/ Five year)
A : total image data (Mbyte).

TABLE IV.
Parameters of the 8-tuple examination_type and four additional parameters acquired at the radiology department of the University Hospital Brussels (VUB) in 1991. Modality clusters can easily be distinguished.

M	E	N	S	R	C	T	G	F	A
CT	Body	34	21	512	12	30	17	2	34
	Head	20	15	512	12	30	10	2	20
MR	Body	80	13	256	12	40	10	1	10
	Neuro	128	13	256	12	40	16	1	16
	Bone	173	25	256	12	55	21.6	1	21.6
	Cardio	19	19	256	12	35	2.3	1	2.3
US	Body	8	4	256	8	20	0.5	3	1.5
DR	Body	8	8	4096	12	15	256	2	500
DSA	Cardio	400	20	512	12	20	200	1	200
	Cardio	100	20	1024	12	30	200	1	200
	Vasc	400	20	512	12	30	200	1	200
PET	Neuro	20	5	256	8	40	1.25	2	2.5
	Cardio	20	5	256	8	40	1.25	2	2.5

8.3 Networking

Broadband communication, includes not only high-speed data transfer via broadband networks but also television by way of terrestrial transmitters, satellites, and cables, and moreover video conferencing and smart building networks. New possible forms of broadband communications are emerging, table V gives examples of relations between the applications and services.

Table V: Broadband Applications and Services for imaging and medical imaging.

Broadband Applications		**Broadband Services**
Category	Examples	
Interactive data communication	PACS CAD/CAM, file transfer, remote printing of newspapers, interconnection of LAN/PABX, access to existing networks/services Realization of MAN (virtual private networks) Colour telefax, browsing through document archives	Data transfer (connection-oriented) Data transfer (conncetionless) Document transfer and retrieval
Interactive video communication	Tele-medicine / tele radiology Group-to-group or person-to-group video communication (point-to-point or multipoint), joint editing Person-to-person video communication, transfer or mail of individual video scenes, video surveillance Information retrieval, teleshopping, counselling, learning, games, video-on-demand	Videoconferencing Video telephony Broadband viedotext, video retrieval
Distibutive communication	Home care/ preventive medicine / medical education TV with today's or enhanced quality (PAL, SECAM, NTSC, or D2-MAC), pay TC, coporate TV TV with high-resolution quality, 3D television	TV distribution HDTV distribution

A distinction must be drawn between applications and services. Applications describe the type of telecommunications used in the subscriber area (including the terminal equipment and the organizational environment). Services are made available on the network in the form of their attributes by the service providers. They are each intended to support a wide range of applications like indicated in table VI.

The applications in the professional sector for broadband technology range from medical image management to computer-aided design (CAD) and computer-aided manufacturing (CAM), and from remote printing of newspapers to colour telefax and workplace videoconferencing, to retrieval of information including pictures, and to electronic cinema, respectively. Group communication within closed user groups or with the aid of virtual private networks in the public network is acquiring increasing importance. Virtual private networks of this kind can be realized, for example, as metropolitan area networks (MANs) on the basis of connectionless data transfer services in the public network. There is a strongly growing demand for this service, with a simultaneous increase in the bit rates being transferred. Also growing in importance is multi-media communication, which is the combined use of data, text, graphics, image, motion video, and audio (voice and sound). Multi-media communication best adapts itself to the human being's complex perception, communication, and way of acting, but also creates new technical demands.

TABLE VI. Communication clusters and there application scenario

Branch	Application Scenario	Benefit
Medicine	Access to patients' case histories/radiographs and specialist advice for operations	* Management and processing * Fast information * in-out hospital
Public administration	Central specialists counselling on the labour market by employment offices and associations	* specific counselling * Higher efficiency
Publishing	Joint editing and remote printing of newspapers and magazines	* Later copy deadline * Lower transport costs
Banking	Electronic branch office	* Comprehensive 24-hour service * Improved security
Industry	Distributed microchip design between the R&D centres of international companies	* Fewer harmonization problems * Short development times
Construction	Planning dialogue between architects/engineers	* Plans always up-to-date * Three-dimensional images
Trade	Maintenance support and training for automotive workshops by a service centre	* Latest know-how * "Natural demonstration"
Home	Teleworking, remote instruction, and teleshopping	* More time/convenience * Greater flexibility

Table VI also indicates the benefits of broadband applications for a series of uses in various branches. The prehensive, more exact, and faster information than was previously possible. For broadband communication it is not suitable to assess future demand by extrapolating from the past. Many of the relevant, influential factors change, and technical solutions are forever in competition, one substituting or superseding another. One example of a comparable development is the widespread use of personal computers, the extent of which could not have been practically predicted just a few years ago. Critical mass and selfstimulation play an important role here.

8.4 Communication needs

Network services: Communication networks are becoming more and more complex and computer networks in particular are becoming more and more heterogeneous. Careful management of these networks becomes a crucial factor for maintaining efficient and reliable network operation and for increasing performance of the quality and the quantity of the services provided to the end-users. Since traditional methodologies for network management are not coping well enough with todays' complex networks, new approaches are being investigated. Recently the utilisation of Artificial Intelligence and Expert System applications has begun to be explored for network management.

Figure 3. shows an possible configuration of network management or PACS service management.

Network management is needed in such an environment, the size of a data transfer being several orders of magnitude larger than in a data transfer in a local area computer network. The existing local area network technology uses protocols which are not suited for image transport. Therefore, it is necessary to sequentialise the image traffic on the network explicitly, i.e. access conflicts are not left to be resolved by the transport layer but are arbitrated by a software module. In addition, the network traffic can be distributed over time because knowledge about patient-, radiologist-, and imageflow is available.

This knowledge can be exploited for careful planning and scheduling like for example organising prefetches of images from archive to local store overnight.

A graphical editor allows the interactive specification of a specific network configuration, i.e. a composition of network segments with associated image acquisition stations, viewing stations, storages, hardcopy facilities, etc. The combination of the generated network model and the simulator allows us to test and experiment with different traffic management strategies and different network configurations.

Traffic management is a prime issue. To a certain point, traffic management can be static, taking into account the network configuration and knowledge of image flow, patient flow and radiologist flow in a hospital environment. But beyond that point decisions on which transfers should be served first, must be made dynamically. They should take into account the current state of network including pending request, actual disk usage, etc.

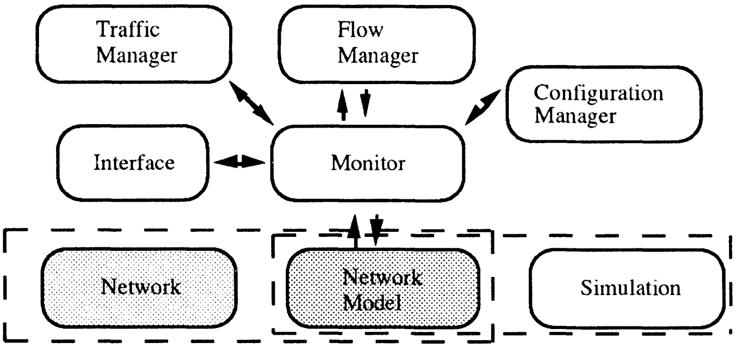


Fig 3. The network management tool; Two main modules of the network management system implement the traffic management task. The flow manager creates and alters image transfers while the traffic manager sequentialises those image transfers over the network.

The next process takes into account the physical dimensions, such as the number of storage layers, the access time of layer and the throughput of the network(s). It will use rules that direct the optimum use of the PACS configuration. A change in the configuration, e.g. a workstation out of duty will be recorded in this data entity. The actual occupancy of the storage and the actual network traffic is recorded in the entity PACS network/storage.

Somewhere the logical description has to be translated into a physical description. It is assumed that this takes place at the start of this process. After the physical routing of an image (moving a certain image from storage x to storage y at a certain point in time) has been decided, the process continues with the physical transport: e.g. breaking the images down into transportable packets. The result is the transfer of an image to another storage (or the release of storage space occupied by an image)

High Connectivity: A source may need to transmit to any one or more of many destinations. A destination (i.e., user) may need to examine many sources.

High Information Rate: Wide-band channels are capable of transmitting many bits per second. The criteria for "high", "wide", and "many" vary widely with type of signal and level of expectation. The 160 Mbits/s information rate channel may seem a high information rate for ordinary interactive computing, but it is too low for convenient transmission of large files (e.g., high-resolution x-rays).

Security-Privacy: Security is a complex of characteristics, some of which provide the technical basis for the protection of privacy. These are two issues that should get special attention in the context of medical computing.

Authentication: A good authentication scheme provides the electronic equivalent of a signature. Ideally, authentication identifies the author of a document and makes it impossible for him to escape responsibility for the authorship. Ideally, also, authentication makes it impossible for anyone to change even one character or bit of the document without destroying the "signature".

High Reliability: Low probability that network service, as seen by the application, will be impaired by malfunctions. For PACS services a full reliability is needed. The PACS architecture should contribute to this.

Priority Service: Guarantied or preferential service, especially when the network is congested, is widely regarded as essential for certain very important functions or for certain very urgent clinical cases.

8.5 Storage media

Database: the kernel of PACS. As databases grow, they put heavy demands on storage capacity.Terabyte systems are likely to be needed in future large image databases. Multi-media DB emerge as a natural evolution of the evolution of the distributed database systems (DDBSs). The main aim of a multi-media DBS (MMDBS) is to provide uniform representation of heterogeneous data in multi-media from multiple types of media. The distributed aspects of such systems are clearly desirable in application areas. In the real world information is often communicated and exchanged among people by means of a variety of media. They communicate through speech, drawing diagrams and pictures, write notes and point things. Therefore in addition to record-based information, people seek computer-based support for storing and managing data represented as voice, images, text, and for communicating using these media.

Load factors are the sum of storage requirements, the level at which each will be stored for what length of time. This sum, in an economic sense, drives the system. It dictates how much hardware will be needed and will usually establish a range in the likely number of workstations and the scope of network requirements. To be sure, processing requirements and constraint factors will exert a major influence on the exact numbers, but in practice most installations fall into patterns. A project that holds only 20 thousand records will not likely require one thousand workstations; however, one that consumes 100 million records will seldom operate efficiently with 10 of those workstations. Load factors are best understood from the three common methods of storage. Not all systems use all three levels, and a few depend only on on-line optical disks.

Magnetic disks: The magnetic disk drives used to store ordinary data work equally well for image systems, provided that they have the capacity to store the much longer records. In practice, they don't have this capacity, nor need they. The fast response time of these drives is necessary only for the most intense processing periods, which usually occur during the first few hours or days after a document is acquired. Thus, only a tiny fraction of the records in an image system will be stored on this medium, and some installations dispense with it entirely.

On-line optical disks: Access time from optical disks is noticeably slower than from magnetic storage, ranging from one to 15 seconds. The reason is that most optical disk libraries are modeled after jukeboxes. A call for a specific records is checked against an index (which itself is almost always kept on a magnetic disk). When the optical disk on which the record is located is identified, a signal is sent to the jukebox to retrieve that disk and insert it into a driver.

Off-line optical disks: This method varies from on-line storage. If the index indicates the disk is no longer on line, a message is sent to an operator who then retrieves the disk from a shelf and inserts it manually in a drive. Depending on staffing and loads, this could take anywhere from a few minutes to several hours, but the images relegated to this level of storage are rarely called for.

In one approach to the trade-off style, all records are kept on-line on optical disks and then the ideal trade-off point for moving them to off-line status is determined. This is usually measured in predictable number-of-accesses per baseline number of records per unit of time, for example, two accesses per 10 thousand records per day. Almost without exception, access requirements are lowered with the passage of time; however, the rate varies with the installation. Then, if extraordinary efficiency is required during initial processing, newly acquired records should be held in temporary magnetic storage. Notice that doubling the time doubles the tax on disk drives, and this could prove expensive.

8.6 Display - Imaging computers

For PACS environments in clinical use, different classes of requirement-levels for image manipulation and viewing can be defined from a medical, organizational, technical and economical point of view. These requirements have a direct impact on the specifications and cost of the image computers as well as on their level of integration in the PACS, RIS (Radiology Information System) and HIS (Hospital Information System) environments. Digital imaging used in radiology requires a high quality display console for the medical specialists to review and manipulate images. Based on the application scenario different classes of stations can be concidered, going from a specialized station to a remote viewing station. These stations can be very advanced platforms based on parallel computing for dedicated image processing functions or advanced multi-media display stations. Only a few common functions are needed on the display stations, like centre and window (gray level manipulation), zooming and panning. The user interface is as in a lot of applications a key factor.

9. The first generation

The first generation of PACS implementations, as developed in the last decade, was mainly technically driven. This proved an important source for the acquisition of experience. It is however generally accepted that they failed to demonstrate a real utility in the clinical environment. As a consequence, PACS have not yet universally been recognized as a useful medical system by all the actors of the health care community.

This failure was partially due to the under estimation of clinical needs, and the lack of maturity of the technical tools (storage, network, digital radiology), their integration and adaptation in the medical world, and to the absence of recognized standards.

Most first generation systems were inadequately integrated within the hospital or radiological information system. This led to inconsistencies in patient data, difficulties and errors in the retrieval of the information. Experience also showed that the need for redundant entering of the patient data via keyboard is a factor of complication; it can generate errors and non-acceptance of the system in the clinical environment.

Finally the problems of the user-interfacing was insufficiently researched and addressed, the needs of the various classes of users are different and specific. Those very requirements cannot be addressed in the centralised architecture.

10. Medical driven approach

A synthesis between the knowledge acquired from the first implementations, the progress of the available technologies, and the cooperative work undertaken by the partners of our research consortium within the AIM exploratory phase (Advanced Informatics in Medicine; an EC funded programme), have led to a innovative concept of the future of PACS.

It can be presented as an integrated but decentralized structure, where small or medium size units are specialized to suit the requirements of specific clusters of users in the hospital. They are linked together as a logical whole that remains transparent for the users.

Integration with other systems in the department. The implementation of a full scale PACS, can only be realized successfully if such a PACS is coupled with the HIS/RIS in that hospital, and this for different reasons.

The images within the PACS need to be identified by means of patient data. To avoid retyping and errors in patient identification, data must be transferred to the PACS can be used in the PACS data-base. The medical history and the radio-diagnostical reports of the patient residing in the HIS must be available on the workstation of the PACS too. It should be possible to present both the image and the alpha-numeric data for the same patient in a coherent way on the PACS workstation. In such a situation, the HIS/RIS functionality, e.g. report editing, should be available through the PACS workstation.

The memory and communication capacities of PACS systems are limited, even using leading edge information technology. To obtain acceptable response times it will be necessary to devise intelligent image management strategies, e.g. to send images in advance from the central archive to the workstation where the user is going to require them (prefetching). Such strategies for image migration can be based on data from the HIS, such as patient location and medical history.

Reporting/consulting stations will provide the interface between fully digital Picture Archiving and Communicating Systems and their clinical users (both radiologists and clinicians). User-friendly, appropriate interfaces which meet the requirements of various user groups and which support the relevant diagnostic tasks will be of utmost importance for the success of PACS.
The adaptive user interfaces relate their functioning to the context they work in. To achieve context sensitivity and reactions, they have to be based on knowledge about the specific requirements of users and tasks and about the general task domain. These explicit and formal user-, task- and domain models may be pre-specified (e.g. using a priori knowledge of the medical diagnostic process, derived from interviews, questionnaires etc.) or acquired and updated during the users' dialogues. The purpose of adaptation and the degree of adaptivity may be defined in wide limits.
User-friendliness is generally regarded to result from a predominantly user-centered, cyclic and interactive design process. User-centered design, however often focuses on some kind of average user and assumes a number of general goals the user is interested in. Changing requirements that arise from different kinds of users and the diversity of goal oriented tasks and sub-tasks which users have to perform are rarely considered during system design and almost never at system runtime. Most human-machine interfaces appear to be static: they function uniformly, regardless of what kind of user is working with a system and what actual task the user is concerned with. Adaptive user interfaces, on the contrary, are dynamic.

Conceptually adaptive user interfaces take into account that workstation users and their actual tasks are variable during runtime. In radiology, digital diagnostic display systems will be used by quite different stereotype categories of users (e.g. radiologists, clinicians, technicians etc.) with diverging duties and information requirements. Even within a particular user group, personal preferences and styles can be observed (e.g. how to arrange images on a display screen). Analysing the filmbased diagnostic process, various categories of goal-oriented tasks can be distinguished, ranging from simple control actions (e.g. image quality check after an examination) to quite complex image handling and display procedures (like preparing a demonstration session for referring clinicians).

Adaptive user interfaces are designed with the main objective to increase usability and convenience. From the user point of view, adaptive systems should appear to be supportive by acting in a cooperative manner. This has to be seen under the boundary condition that consistency and predictivity of the interface will not be affected by the adaptation mechanisms. From the designers' point of view, user interfaces have to be based on formalized descriptions (model) of the requirements for different users and tasks. These models have to define major parts of the workstations' functionality on a high level of abstraction.

11 Open systems

From a pragmatic viewpoint, the development of an open system presents many of the same challenges encountered in the development of any complex software system-concern with reliability, maintainability of software over a long timeframe. An open system, however, also brings unique problems: for example, it may embody several different computer systems, comprised of different hardware and different operating systems, may involve several independent databases, and it may be spread over a wide geographical area. In the specification and design of an open system one may have to deal with existing databases, foregoing the luxury of building from scratch more convenient specifications. Moreover, because an open system involves processes executing on several machines, the testing of an open system is difficult, and often requires the development of specialized tools. PACS / IMACS environment should be considered as open systems in this multi-vendor world.

12. Standardization

In order to respond to the challenge of integration and communication in the world of Medical Informatics, in June 1990 the European Standardization Committee (CEN) founded a new Technical Committee: TC251 "Medical Informatics". CEN TC251 is responsible for the standardization activities in Medical Informatics in Europe. CEN TC251 deals with the organization, coordination and monitoring of the development of standards, including testing standards, in Health Care Informatics as well as the promulgation of these standards. Seven different working group are created.

1.healthcare information modeling and medical records
2.healthcare terminology, semantics and knowledge bases
3.healthcare communications and messages
4.medical imaging and multi media
5.medical devices
6.healthcare security and privacy, quality and safety
7.intermittently connected devices

Different items are defined for Standardization work in the area of Medical Imaging and Multi Media. Some short-time targets and needs are defined, other are more long-term and topdown,

-Functional profiles for medical image interchange
-Medical Image management standards
-Medical Image and Related data interchange Format Standards
-Off-line media
-Standard classification and codes for Medical Image Processing

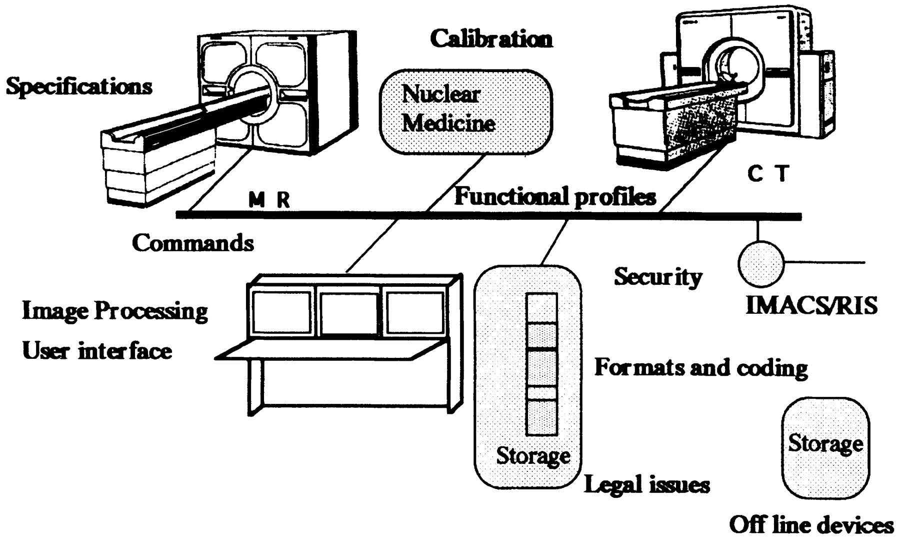


Fig 4. A possible PACS / IMACS architecture, with bold indication the standardization needs.

The first items were selected by CEN TC251 as very important and work has started in these areas. Functional profiles are given to EWOS in close cooperation with WG4 because this item is related to open system protocols which is the speciality of EWOS. For the other, project teams are set up. First draft proposals are planned for the end of 1992. Figure 4 shows an possible PACS configuration. Bold words indicate standardization needs.

Of course close contacts exist between the international activities and one international standard is also one of the concerns of the working groups. Standardization is needed for interoperability between these heterogeneous systems, and it will also stimulate the technology and communication systems. These are necessary services for the information systems like PACS. These will then increase the usage of IT&T in the healthcare sector, resulting lower price setting in this complex domain and easier integration of the new communication components. These interaction is clearly reflected in the figure 5.

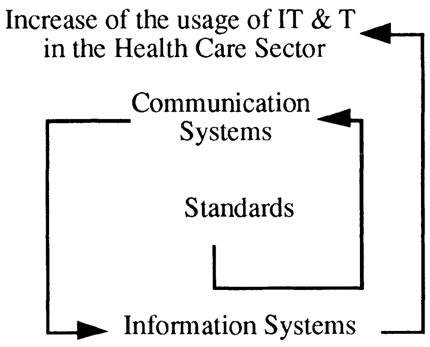


Fig 5. The Spiral phenomenon; In the health care sector the use of IT&T will increase is standards are defined. Standards are needed for communication systems, which are the driving fores for new generation information systems.

13. Conclusion: Film Independent Radiology

The Health care model information flow clearly indicated the needs for communication between the medical doctor and specialist, between specialist and services providers and inside, between the service provider and the resources.

The main technical components of an PACS / IMAC world are given, starting with the image acquisition units. These devices in one or another way generate the images, but also these devices also need to be connected to the communication and network environment. A distributed data-base and hierarchical storage level are necessary for management of the images, while special display units will be needed for the high resolution images. More and more direct digital acquisition units are installed in the hospitals and new ones like digital Radiology based on phosphor plates are coming soon.

Different components important in the health care model evolution process were described. A discussion is given of hardware, software, communications, data and users. An analogy is found with open systems and in the same line an extraordinary range of tools and services are needed, a precondition to the success of such an environment will be a revolutionary change in the technology of user interfaces. Components coming from the Information Technology & Telecommunication market should be used from a technical and economical point of view. Standardization is a high-priority need if we don't want to end in a total communication chaos. Present image generation and image management devices or IMACS components have no standardised transport systems and are not designed to operate in a heterogeous multi-vendor environment. Standardization will play an important role and is a must for this medical informatics area.

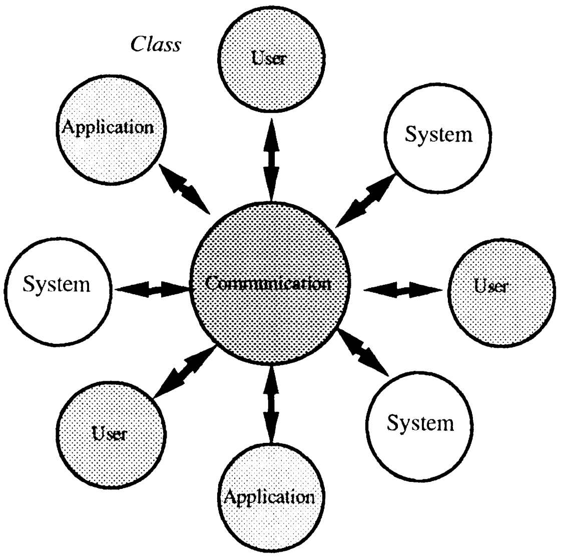


Fig 6. Communication is the kernel of the PACS/IMACS service: Communication between user, systems and processes.

Short time and long term targets are needed, and here, too standards from the IT&T market should be used if appropriate.

Tele-medicine will be an important area for Europe, services are needed to ensure a step by step approach, integrating technology from different fields.

A PACS service is needed not because we want a film less radiology department but a film independent hospital and want to add additional possibilities to image manipulation, communication and processing.

The back-bone of these IMAC's is of course "communication"; like indicated in figure 6. Communication means integration. Communication between pieces of equipment, communication between application sessions and communication with users. Successful implementation of IMAC does not end with either the development of a good system of the implementation of a series of pieces of equipment. IMAC means new ways of doing business and changes in organizations caused by changes in information flow patterns. Operational scenarios in the network environment must be considered from the beginning.

Elimination of film cannot solve all the problems in image management. It will on the other hand create a number of new problems. Total elimination of films will be difficult to achieve in a large medical centre for many years to come, because the films will be required by other non-IMAC hospitals, but it will be possible to develop a film-independent imaging service in the near future.

IMAC is inevitable and necessary. It will affect the way radiology is practiced. Changes in the pattern of information flow will bring about changes in work patterns and changes in people's ability to exercise professional expertise. These changes must come from prudent and deliberate planning by the user community rather than from external organizations. IMAC is "my problem" for everyone and affects many people in health care organizations. Everyone has different and often competing interests. As stated earlier, changes in information flow bring about changes in power that may produce strong reactions. IMAC technology may intimidate poorly informed parties.

References

Boehme J.M. , Chimiak W. , Choplin R., Maynard C.D. ; 1991," Operational Infrastructure for a Clinical Picture Archiving and Communication System" *SPIE Medical Imaging v: PACS Design and Evaluation* , vol. 1446, pp. 312-317.

Huang H.K., Ratib O.,Bakker A.R.,Witte G.,1991," Picture Archiving and Communication Systems in Medicne", *NATO ASI Series*, Springer Verlag. , Vol F74

Mattheus R. ,Temmerman Y.,Verhellen P , Osteaux M. ; 1991" Management system for a PACS network in a hospital environment", *SPIE Medical Imaging v: PACS Design and Evaluation* , vol. 1446, pp. 341-351.

Mattheus R.; " Standardization in medical imaging, 1991," *Health Technology Standards*, S. W. Gunn, N.J. O'Riordan, Eds.,IEC/ISO Switzerland. pp. 541-549.

Mattheus R, Moyson F, Temmerman Y , Osteaux M , 1990, Hospital Integrated Picture Archiving and Communication System (HIPACS):A European Project; Symposia Foundation SCAR. 396-404.

Mun S.K., Beason H., Elliot L.P., Göhringer F., Saarinen A., Haynor D, 1989, "Total digital department: implementation strategy". *Proceedings Medical Imaging III, SPIE* Vol. 1093, pp. 133-139.

Noothoven van Goor J., Pihlkjaer Christenses J..; editors, 1992, *Advances in Medical Informatics*;.Amsterdam, IOS Press.

Osteaux M., Bakker A., Bell D., Kofakis P., Mattheus R., Meyer-Ebrecht D., Van de Velde R. ,Wendler Th., 1992; " Hospital Integrated Picture Archiving and Communication Systems "; Springer-Verlag; ISBN 3-540-54592-1,