



IEE DIGITAL ELECTRONICS AND COMPUTING SERIES 5

Distributed computing systems programme

Edited by D.A.Duce

Distributed computing systems programme

Edited by D.A.Duce

Peter Peregrinus Ltd
On behalf of The Institution of Electrical Engineers

Published by: Peter Peregrinus Ltd., London, UK.

© 1984: Peter Peregrinus Ltd.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means — electronic, mechanical, photocopying, recording or otherwise — without the prior written permission of the publisher.

While the author and the publishers believe that the information and guidance given in this work is correct, all parties must rely upon their own skill and judgment when making use of it. Neither the author nor the publishers assume any liability to anyone for any loss or damage caused the result of negligence or any other cause. Any and all such liability is disclaimed.

British Library Cataloguing in Publication Data

Distributed computing systems programme —
(IEE digital electronics and computing series 5)
1. Management — Data processing
2. Electronic data processing — Distributed
processing
I. Duce, D.A. II. Series
001.64 HF5548.2

ISBN 0-86341-023-5

Printed in England by Short Run Press Ltd., Exeter

Contents

Preface	xi
List of contributors	xiii
1 The Distributed Computing Systems Programme 1977-1984	1
1.1 BACKGROUND	1
1.2 MANAGEMENT OF DCS	3
1.3 INFRASTRUCTURE ACTIVITIES	4
1.3.1 Meetings programme	5
1.3.2 Conferences	5
1.3.3 Mailshot	5
1.3.4 Annual Report	5
1.3.5 Equipment pool	5
1.3.6 Technical Backup	6
1.4 RESEARCH THEMES	6
1.5 INDIVIDUAL PROJECTS	8
1.6 SOME STATISTICS	9
1.7 ACHIEVEMENTS	9
ACKNOWLEDGEMENTS	10
REFERENCES	11
2 The Sussex Broadband LAN Project	12
2.1 INTRODUCTION	12
2.2 REQUIREMENTS	12
2.3 BROADBAND DATA NETWORKS	14
2.4 THE SUSSEX NETWORK	14
2.4.1 Low Bit Rate Fixed Frequency Modem	17
2.4.2 Frequency Agile Modem	19
2.4.3 High Bit Rate Modem	20
2.5 CURRENT STATUS AND DISCUSSION	21
3 Implementation of a High Performance LAN - Centrenet	25
3.1 INTRODUCTION	25
3.2 CENTRENET PHILOSOPHY	26
3.3 CENTRENET ARCHITECTURE	27
3.4 MARK-1 STARPOINT DESIGN	30
3.5 MARK-2 PORT-CARD	31
3.6 THE SUPERPORT	32
3.7 REMOTE LINKS	33
3.8 LOCAL LINKS	35
3.9 CONCLUSIONS	36
ACKNOWLEDGEMENTS	37
REFERENCES	37
4 Imperative Languages in Distributed Computing	39
4.1 INTRODUCTION	39
4.1.1 Program structuring	40

vi Contents

4.1.2	Communication mechanisms	41
4.1.3	Making connections	42
4.1.4	Non-determinism	43
4.2	IMPLEMENTATION INFLUENCES	44
4.2.1	Timing problems	44
4.2.2	Hardware failure	46
4.2.3	Shared memory	46
4.3	THE LANGUAGES	46
4.3.1	Ada and Martlet	47
4.3.2	Conic	50
4.3.3	occam	53
4.3.4	Pascal-m	55
4.3.5	Path Pascal and PascalPlus	56
4.3.6	Programming in unhelpful languages	57
4.3.7	Other languages	58
4.4	CONCLUSION	59
	REFERENCES	59
5	A Strongly Typed, Distributed Virtual Memory	62
5.1	INTRODUCTION	62
5.2	FACILITIES REQUIRED	64
5.2.1	Types supported	64
5.2.2	Manipulation from within a program	65
5.2.3	Management of the Structured Data Store	65
5.2.4	Representations	66
5.2.5	Operations	68
5.2.6	Distributed Data	68
5.2.7	Requirements Summary	69
5.3	THE VIRTUAL MEMORY	69
5.3.1	Data Structure Accessing Operations	70
5.4	THE USER INTERFACE	72
5.4.1	The Type Editor	72
5.4.2	The Value Editor	79
5.5	CONCLUSION	81
	REFERENCES	83
	ACKNOWLEDGEMENTS	85
6	Building Flexible Distributed Systems in Conic	86
6.1	FLEXIBILITY IN DISTRIBUTED SYSTEMS	86
6.2	CONIC MODULE PROGRAMMING LANGUAGE	88
6.2.1	Task Modules	88
6.2.2	Communication Primitives	90
6.2.3	Input/Output	91
6.3	CONIC CONFIGURATION LANGUAGE	92
6.3.1	Context Definition	92
6.3.2	Instantiation	93
6.3.3	Interconnection	93
6.3.4	Mapping onto Physical Topology	94
6.3.5	Structuring Configuration Specifications	94
6.4	DYNAMIC CONFIGURATION	96
6.4.1	Change Specifications	97
6.4.2	Configuration Manager	97
6.5	DISTRIBUTED OPERATING SYSTEM	99
6.5.1	Station Executive	100
6.5.2	Utilities	101
6.5.3	Configuration Operations	102
6.5.4	Performance	102
6.6	CONCLUSIONS	103
6.6.1	Experience of Using Conic	103

	6.6.2	Current Status	104
	6.6.3	Future Work	104
		ACKNOWLEDGEMENTS	105
		REFERENCES	105
7		The Cosy Approach to Distributed Computing Systems	107
	7.1	CONCURRENT, DISTRIBUTED AND SYNCHRONIZED SYSTEMS	107
	7.2	DECISIONS INFLUENCING THE DESIGN OF THE COSY MODEL	108
	7.3	TOWARDS THE COSY MODEL	109
	7.4	THE COSY MODEL	111
	7.5	FORMAL RESULTS ABOUT THE MODEL	116
	7.5.1	Static Criteria for Adequacy and Periodicity	116
	7.5.2	Equivalence of COSY Specifications	116
	7.5.3	Mathematical Results about Vector Firing Sequences	116
	7.6	TRANSFORMATIONAL DEVELOPMENT OF SPECIFICATIONS	117
	7.6.1	Constrained Expansion and Reduction Rules	117
	7.6.2	Decomposition of Sequential into Concurrent Systems	117
	7.7	DEVELOPMENT OF A COMPUTER BASED ENVIRONMENT FOR COSY	118
	7.7.1	The Basic COSY System : BCS	118
	7.7.2	The COSY Dossier	118
	7.8	HIGH LEVEL COSY NOTATION	118
	7.9	APPLICATIONS OF COSY	119
	7.9.1	Operating System Problems	119
	7.9.2	Network Protocols	119
	7.9.3	Train Journeys	119
	7.10	VLSI IMPLEMENTATION OF COSY	119
	7.11	RELATION OF COSY MODEL TO OTHER APPROACHES	119
	7.11.1	Other semantic models	120
	7.11.2	Programming Notations	122
		ACKNOWLEDGEMENTS	122
		REFERENCES	122
8		Ease of Use Through Proper Specification	126
	8.1	INTRODUCTION	126
	8.2	A FIRST EXAMPLE	126
	8.3	THE FIRST COMPROMISES	128
	8.4	A COMPROMISE AVOIDED	131
	8.5	MODULARITY AND COMPOSITION OF SERVICES	134
	8.6	EXPERIENCE SO FAR	136
	8.7	FUTURE PLANS	137
	8.8	GLOSSARY OF SYMBOLS	137
		REFERENCES	138
9		Probabilistic Modelling of Distributed Computer Systems	139
	9.1	GENERAL BACKGROUND	139
	9.2	MODELS THAT CAN BE SOLVED EXACTLY	141
	9.2.1	Multiprocessor Systems with a Single Job Type	142
	9.2.2	Two Job Types	143
	9.2.3	Network Models	144
	9.3	MODELS THAT REQUIRE APPROXIMATIONS	147
	9.3.1	A Distributed Data Base Model	148

viii Contents

9.3.2	Local Area Networks	149
9.4	CONCLUSION	152
	REFERENCES	152
10	Developing Concurrent Systems with DTL	154
10.1	INTRODUCTION	154
10.2	PROCESS ABSTRACTION	155
10.3	CONCURRENT TRANSLATIONS	156
10.4	EXAMPLE : SORTING	157
10.5	SEQUENTIAL TRANSLATIONS	159
10.6	EXAMPLE : LOGIC NETWORKS	161
10.7	IMPLEMENTATION OF DTL PROGRAMS	165
10.8	SUMMARY	166
	REFERENCES	166
	ACKNOWLEDGEMENTS	168
11	Parallel Algorithm Design	169
11.1	INTRODUCTION	169
11.2	NUMERICAL PARALLEL ALGORITHMS	169
11.2.1	Algorithm Structure	170
11.2.2	Parallel Methods for the Tridiagonal Eigenvalue Problem	171
11.2.3	Analysis of the Alternative Solution Methods	173
11.2.4	Results	175
11.3	NON-NUMERICAL PARALLEL ALGORITHMS	176
11.3.1	Introduction	176
11.3.2	The Neighbour Sort Algorithm	176
11.3.3	The Parallel 2-Way Merge Algorithm	176
11.3.4	The Complexity of the Algorithms	176
11.4	CONVERSION OF IMPLICIT METHODS TO EXPLICIT FORM	181
11.4.1	A New Group Explicit Method	183
11.5	NEW PARALLEL ALGORITHMS	185
	ACKNOWLEDGEMENTS	185
	REFERENCES	185
12	Design Study for Active Memory Arrays	187
12.1	DESIGN AIMS	187
12.2	RELATED WORK	189
12.2.1	High integrity design	190
12.2.2	Microcoding	191
12.2.3	Parallel arithmetic	191
12.3	PN SYSTEM	192
12.3.1	Abstraction	192
12.3.2	Memory management	193
12.3.3	Array manipulation	194
12.3.4	Instruction sequencing	195
12.4	PERFORMANCE	196
	ACKNOWLEDGEMENTS	198
	REFERENCES	198
13	Hardware and Software for Parallel Update of Raster Graphics Images	199
13.1	INTRODUCTION	199
13.2	A QUICK INTRODUCTION TO RASTEROP	200
13.3	DISPLAY LIST APPROACH	201
13.4	DISARRAY, RASTEROP AND LINE DRAWING	203
13.4.1	Rasterop	203
13.4.2	Line Drawing	204
13.5	A SIMPLE EXAMPLE	204
13.6	THE ARRAY PROCESSOR	207

13.6.1	Overview	207
13.6.2	The Processing Elements	209
13.6.3	The Basic Array Cycle	211
13.6.4	The Array Control Unit	211
13.6.5	The Refresh Controller	211
13.7	ADDRESS STAGGERING SCHEME	212
13.8	QUADRANT ADDRESSING SCHEME	213
13.9	DISARRAY PERFORMANCE	214
13.10	DISARRAY2	215
13.10.1	The Next Generation	215
13.10.2	Surface Shifting	215
13.10.3	Processor Element Implementation	217
13.10.4	Other Disarray2 Features	217
13.11	CONCLUSIONS	218
REFERENCES		219
Directions in Functional Programming Research		220
14.1	INTRODUCTION	220
14.2	LANGUAGES	221
14.2.1	Polymorphic Typing	222
14.2.2	Syntax	222
14.2.3	Modules and Abstract Data Types	224
14.2.4	Functional and Logic Programming	224
14.2.5	Debugging	225
14.3	PROGRAM TRANSFORMATION	225
14.3.1	Correctness	226
14.3.2	Transformation Systems	227
14.3.3	Specification Languages	228
14.4	EVALUATION ORDER	228
14.4.1	Semantic Issues	228
14.4.2	Pragmatic Issues	232
14.5	IMPLEMENTATIONS	237
14.5.1	Reduction Machine Models	237
14.5.2	Memory System and Garbage Collection	238
ACKNOWLEDGEMENTS		241
REFERENCES		242
15	The Zero Assignment Parallel Processor (ZAPP) Project	250
15.1	INTRODUCTION	250
15.2	ARCHITECTURE RESULTS	251
15.2.1	The Reduction Model of Computation	251
15.2.2	Process Trees	252
15.2.3	ZAPP Architectures	254
15.2.4	A General Model of Divide and Conquer Algorithms	255
15.2.5	Using a ZAPP	256
15.2.6	General Behaviour of a Single-user ZAPP	257
15.2.7	Experiments	258
15.3	THEORETICAL RESULTS	260
15.3.1	Rewrite Systems	260
15.3.2	Graph Rewriting on ZAPP	261
15.3.3	A Language for Expressing Parallel Graph Reduction Algorithms	263
15.4	CONCLUSIONS	266
ACKNOWLEDGEMENTS		267
REFERENCES		267
16	The Manchester Dataflow Project	270
16.1	INTRODUCTION	270
16.1.1	A Brief History of the Manchester Project	270

16.1.2	Tagged-Token Dataflow Computation	272
16.2	ENGINEERING CONSTRUCTION	274
16.2.1	System Architecture	275
16.2.2	Physical Construction	276
16.2.3	The Inter-Module Interface	276
16.2.4	CAD Tools	277
16.2.5	The Matching Unit	277
16.2.6	The Structure Store	278
16.2.7	Multi-Ring Systems	278
16.3	LANGUAGE SYSTEMS	279
16.3.1	PO and Pascal	279
16.3.2	Lapse and Mad	280
16.3.3	Lucid and SASL	280
16.3.4	DP, CSP and Id Resource Managers	281
16.3.5	Assembly-Level Languages	281
16.3.6	SISAL and Intermediate Format	281
16.4	TEST SYSTEMS	282
16.4.1	Hardware Test Programs	282
16.4.2	Software Fault-Tracing	283
16.4.3	Symbolic Debugging	283
16.5	APPLICATION PROGRAMS	283
16.5.1	Small Programs	283
16.5.2	Physics Calculations	284
16.5.3	Computer-Aided Design	284
16.5.4	Standard Parallel Benchmarks	284
16.5.5	Program Characteristics	284
16.6	PERFORMANCE EVALUATION AND TUNING	285
16.6.1	Preliminary Performance Evaluation	285
16.6.2	Pipeline Tuning	285
16.6.3	Future Work	286
16.7	CONCLUSIONS	287
	ACKNOWLEDGEMENTS	287
	REFERENCES	288
17	Shells of Functional Operating Systems	290
17.1	INTRODUCTION	290
17.2	SIMPLE INTERACTION	290
17.3	SEQUENCES OF INTERACTION	292
17.4	INTERACTION WITH DATABASES	294
17.5	INTERACTING WITH A DATABASE OF PROGRAMS	296
17.6	CONCLUSIONS	297
	REFERENCES	298
	Index	299

Preface

This volume is based on papers presented at the Final Conference of the U.K. Science and Engineering Research Council's coordinated programme of research in Distributed Computing Systems (DCS), held at the University of Sussex, Brighton, U.K. in September 1984.

Chapter 1 explains the origins and history of the DCS Programme and gives an overview of the technical development of the programme, emerging research themes and achievements.

Chapters 2 and 3 are concerned with two approaches to local area networks. The Sussex network is a coaxial cable network using the broadband approach, whereas the University of Manchester's Centrenet is a high bandwidth network using optical fibre as the transmission medium.

Chapter 4 is a review of the major imperative languages (including Ada and variants of Pascal) which have been applied to, or developed for, distributed computing.

Chapter 5 describes a strongly typed, distributed virtual memory developed at UMIST. This paper shows how it is possible to extend the data abstraction facilities of modern programming languages to include all data, both volatile and persistent, local and distributed.

Chapter 6 describes the Conic system, an integrated set of tools for constructing and managing large distributed computer control systems. A key feature of Conic is the ability to manage evolving systems. A system built using Conic can easily incorporate new functionality in response to evolutionary changes and existing components can be reorganized in response to operational changes.

The problem of specification of distributed systems introduced in Chapter 6 is taken up in the next two chapters. Chapter 7 describes the COSY system, a system to support rigorously reasoned design, development and analysis of concurrent systems. Chapter 8 introduces the Z specification technique and explores its application to distributed operating systems.

xii Preface

Chapter 9 is a review of techniques for modelling distributed systems mathematically. Systems discussed include distributed databases and local area networks.

Chapter 10 introduces the DTL language for specifying and developing concurrent systems. Programs are expressed as structured networks of translations which communicate data on fully synchronized streams.

Chapter 11 discusses the design and performance analysis of parallel algorithms. Both numerical (eigenvalue problem, heat-conduction problem) and non-numerical (sorting, merging) problems are explored.

Chapters 12 and 13 describe current research work at Queen Mary's College, University of London. The first presents the active memory array concept for the design of high-performance, high-integrity machines. The second describes a solution to the problem of updating raster graphics images at high speed, based on a regular two-dimensional array of simple processor/ memory pairs.

The remaining chapters are concerned with declarative languages and non-von Neumann computer architectures. Chapter 14 reviews current directions in functional programming research. Topics covered include languages, program transformation, evaluation order and implementations. Chapter 15 describes the ZAPP project, which demonstrates how it is possible to "buy speed" for a significant class of problems by linking together a large number of computing elements. Chapter 16 concerns the Manchester dataflow project. Some of the technical problems encountered in the construction of the prototype dataflow computer are described along with their solutions. Software and performance evaluation issues are also covered. Functional operating systems are discussed in the final chapter, and it is shown how the shell (the component responsible for structuring the interaction with the system user) of an operating system can be built as a composition of purely functional programs.

The preparation of this volume has involved a great deal of work on the part of the contributors, to whom I owe a deep debt of gratitude. This volume also owes much to my secretaries, Janice Gore and Lilian Vallentine for handling the mountain of correspondence that has crossed my desk; and to my colleagues Fred Chambers, Gill Jones and Elizabeth Fielding. Finally, I should like to thank Dick Grimsdale and John Golds at the University of Sussex for hosting the Conference which provided the incentive to produce this volume.

May this volume be a reminder of the enthusiasm and spirit of cooperation which has existed in the DCS Programme and may it serve as a source of encouragement to present and future workers in the field.

Feast of St Peter and St Paul
1984

David A. Duce

List of contributors

Chapter 1

R Newey
Marconi Electronics
7835 East Redfield Road
Scottsdale
Arizona 85260
USA

Chapter 2

F Halsall
School of Engineering and Applied Sciences
University of Sussex
Falmer
BRIGHTON
BN1 9QT

Chapter 3

D A Edwards, R N Ibbett & T P Hopkins
Department of Computer Science
University of Manchester
Oxford Road
MANCHESTER M13 9PL

Chapter 4

R Bornat
Dept of Computer Science & Statistics
Queen Mary College
Mile End Road
LONDON
E1 4NS

Chapter 5 and 10

J W Hughes & M S Powell
Dept of Computation
University of Manchester Institute of Science
& Technology
Sackville Street
MANCHESTER
M60 1QD

Chapter 6

M Sloman, J Magee & J Kramer
Dept of Computing
Imperial College
180 Queen's Gate
LONDON SW7

Chapter 7

P Lauer
Computing Laboratory
University of Newcastle upon Tyne
Claremont Tower
Claremont Road
NEWCASTLE UPON TYNE
NE1 7RU

Chapter 8

R Gimson & C Morgan
Programming Research Group
University of Oxford
8-11 Keble Road
OXFORD
OX1 3QD

Chapter 9

I Mitrani
Company Laboratory
University of Newcastle upon Tyne
Claremont Tower
Claremont Road
NEWCASTLE UPON TYNE
NE1 7RU

Chapter 11

D J Evans
Dept of Computer Studies
Loughborough University of Technology
LOUGHBOROUGH
Leicestershire
LE11 3TU

Chapter 12

J K Iliffe
Dept of Computer Science & Statistics
Queen Mary College
Mile End Road
LONDON
E1 4NS

Chapter 13

I Page
Dept of Computer Science & Statistics
Queen Mary College
Mile End Road
LONDON
E1 4NS

Chapter 14

S L Peyton-Jones
Dept of Computer Science
University College
Gower Street
LONDON
WC1E 6BT

Chapter 15

J R Kennaway & M R Sleep
School of Computing Studies
University of East Anglia
University Village
NORWICH
NO4 8BC

Chapter 16

J R Gurd, C C Kirkham & I Watson
Dept of Computer Science
University of Manchester
Oxford Road
MANCHESTER
M13 9PL

Chapter 17

P Henderson & S B Jones
Dept of Computer Science
University of Stirling
Stirling
FK9 4LA

Chapter 1

The distributed computing systems programme 1977-1984

R. Newey

This volume forms the proceedings of a conference on Distributed Computing, sponsored by the U.K. Science and Engineering Research Council (SERC) at the University of Brighton, Sussex 5/6 September 1984. The conference was the culmination of SERC's Distributed Computing Systems Programme (DCSP), 1977-84. The papers in this volume come from researchers funded by the programme. Some papers describe individual research projects, others review particular areas in the field.

This introductory paper describes the background to the programme, its modus operandi, and technical development.

1.1. BACKGROUND

The body principally responsible for funding computer science research in U.K. Universities and Polytechnics is the SERC. This responsibility is discharged in the main by the awarding of research grants to institutions. Funds may be requested for staff, travel and equipment, and in addition investigators may request access to the council's own facilities, for example computing resources, typically located in one of the council's laboratories. Research grant applications are considered by the council's committees, whose members are drawn from academe and from industry, together with assessors from other government departments. Thus the principle of peer review is adopted, within a context which evaluates the intellectual potential of the proposed research together with the possibility of, eventual, industrial value.

The council operates mainly in a responsive mode, responding to applications submitted by its clients, rather than a directive mode in which the initiative for new projects rests with the council. Prior to 1977 Information Technology was funded purely in this responsive way, although examples of different ways of working could be found in other areas of the council's activities. There had developed a strong feeling within the council committees concerned, that it would greatly enhance the state of computing research in the U.K. if there could be identified a common focus for new activity. At the same time, it was becoming increasingly obvious that a principal barrier to further development of computing systems, was the inability to manage the power becoming available from cheaper, faster hardware; particularly when many processors were grouped in complex, concurrent networks: that is, distributed computing. The Computer Science committee, recognizing the importance of distributed computing as a research area, appointed a panel in June 1976 under the chairmanship of Prof. I. Barron, to consider what action was necessary to encourage, coordinate or direct research in Distributed

2 Distributed computing systems programme

Computing. This panel was asked to take particular account of the potentially high cost of such research and the avoidance of unnecessary duplication of effort.

In its report to the committee in October 1976, the panel recommended that a coordinated research programme should be established and that additional funds should be sought for the programme. When a draft programme was circulated to relevant academic departments for comment, more than 50 replies were received, the great majority expressing a desire to participate and offering useful criticisms of the proposed mechanics and content of the programme. A one-day Workshop, in March 1977, provided an opportunity for a direct exchange of views on the original proposal and on the research problems to be addressed. The panel, in response to these helpful interchanges of views, revised a number of its original proposals.

The eventual proposal to set up a coordinated programme of research into Distributed Computing Systems was warmly welcomed by the Engineering Board of SERC. Approval in principle for the programme was given by the Board in June 1977; the programme was initiated in the academic year 1977-78. DCS was the first attempt by SERC to establish a long term, extensive, coordinated programme of research in Information Technology.

The primary scientific objectives of the programme were to seek an understanding of the principles of Distributed Computing Systems and to establish the engineering techniques necessary to implement such systems efficiently. These broad objectives reflect the relative immaturity of the subject when the programme was founded. In particular the programme sought to establish an understanding of parallelism in information processing systems and to devise ways to take advantage of this.

The practical objectives of the programme can be summarized [1] as: to achieve results of practical value to the U.K. industry by directing research to a key area for the future; to promote relevant Computing Science research of high quality in academic departments by coordinating the efforts and achievements of individual research teams, and to ensure a cost effective research programme.

A Distributed Computing System was considered to be one in which there are a number of autonomous but interacting computers cooperating on a common problem. The essential feature of such a system is that it contains multiple control paths executing different parts of a program and interacting with each other. Such systems might consist of any number of autonomous units, but the more challenging problems involve a large number of units. Thus, the spectrum of Distributed Computing Systems includes networks of conventional computers, systems containing sets of microprocessors, and novel forms of highly parallel computer architecture with greater integration of processing and storage.

The motivations for and importance of research into distributed computing systems are many and varied. Some major ones are:

- Performance: eventually it will be impossible to increase the speed of a single processor and retain commercial viability. Several processors, cooperating on a single task, will be the only way to greatly enhance performance.
- Reliability: a fully distributed system should be able to tolerate faults caused by either software or hardware. Hardware faults can be tolerated for example, by having more than one of each critical element. Software faults can be reduced by running different algorithms in parallel and checking the validity of results.
- Clarity: many problems are naturally parallel. Some problems are inherently simpler if expressed as a set of interconnected and communicating processes. If a problem's solution is expressed in this way it could be easier to provide a proof of correctness for the whole solution by breaking the proof task down, first proving the correctness of individual processes, and then proving the correctness of their interconnection.
- Distribution: in areas such as real time control it is often important that processor power is available where it is required in order to minimise the bandwidth requirements of data paths.
- Cost: the low cost of microprocessors allows certain tasks to be performed more economically on sets of microprocessors than on a single main frame processor.

1.2. MANAGEMENT OF DCS

We have already mentioned two extreme ways in which SERC operates, responsive mode and directed mode. In responsive mode the initiative for new projects rests entirely with investigators. Submissions are made to SERC as new ideas are conceived and support becomes necessary. The committees reviewing grant applications can only exercise control at bottom, by accepting or rejecting applications. The other pole to this model is a totally directed programme in which essentially, the Director, issues invitations to research groups to work on particular problems. The initiative for new work then rests entirely with the Director. In practice, of course, neither extreme is fully adopted - any research programme which is to be successful must take account of the talents and interests of the research community, and equally the community must understand that the availability of limited funds makes it impossible to pursue all lines of research that, at first sight, seem promising; priorities must be established.

Within this scheme, a coordinated programme falls mid-way between the two extremes. The aim of coordination is to establish a symbiotic working relationship between the committee responsible for the management of the programme, whose chief concern lies in the proper administration of limited resources, and the researchers whose concern is the pursuit of knowledge in their chosen disciplines. It is essential to create an atmosphere of mutual understanding and cooperation among the researchers themselves, and to create an environment in which research ideas can be discussed and priorities agreed.

4 Distributed computing systems programme

There were two main reasons for adopting a coordinated approach to research in distributed computing. First reflecting the importance of the subject to the progress of computing and information technology, it helps to ensure a reasonable balance of SERC support across the various areas concerned; and a framework facilitating take-up of research results by industry. Second the substantial costs of much research in this field, and the limits of funds available, make it essential to provide support in a cost effective way - without impinging on the necessary freedom of investigators in carrying out fundamental research.

The idea of a fully directed programme was explored but subsequently rejected. (The idea of a directed I.T. programme has now been realized in the Alvey programme.)

What then are the practicalities of coordination? Coordination has been achieved at two levels. First within the programme itself, there has been continuing contact with investigators, starting with assistance in formulating research proposals, and throughout the research period, by regular interchange of information both spoken and written. Second, outside the programme, links have been fostered with industrial organizations (including government establishments etc.) likely to make use of the research in some form.

The coordination team includes an Academic Coordinator (drawn from the staff of SERC Rutherford Appleton Laboratory), responsible principally for liaison with and monitoring of the research projects; and an Industrial Coordinator, (from industry) who is also charged with expanding the external contact range of the research, with a view to collaborative research and technology transfer. The coordinators are supported by a Technical Secretary and by various support and development staff maintaining and enhancing the programme's infrastructure. When the programme was established it was envisaged that coordination would not be a very demanding activity and so the first coordinators were employed on a part-time (1 day per week) basis. It rapidly became clear that this premise was false and that coordination is a very demanding activity indeed. Since the earliest days of the programme the Academic Coordinator has been employed on a full-time basis, but the Industrial Coordinator has remained a part-time appointment.

The programme has been monitored and controlled by a panel of researchers and industrialists, appointed by the council, to whom the coordinators report. The panel has both evaluated and recommended award of research grants in DCS subject areas, and considered regular six-monthly progress reports from each group in the programme.

1.3. INFRASTRUCTURE ACTIVITIES

A major factor in the development of the DCS programme has been the recognition of the need for a well appointed infrastructure on which to build research programmes. The infrastructure provided by DCS has included a meetings programme of workshops and conferences, a community wide mailshot, Annual Report, an equipment pool, and a high level of backup technical support.

1.3.1. Meetings programme

To be successful, a coordinated programme must engender a sense of community in its participants. It must also bring together disparate groups of researchers in a constructive way, to foster the germination of new ideas or new approaches to traditional problems. The single most influential factor in bringing this about within the DCS programme has been the workshop programme. Some of the most significant pieces of research in the programme can be traced back to particular gatherings of researchers. Considerable experience in organizing workshops has been built up in DCS. The most fruitful meetings have typically been of 1 1/2 days duration and limited to 25-30 participants. As the programme evolved and research themes became established, meetings of researchers in each area were held at regular intervals, to monitor and discuss progress.

1.3.2. Conferences

As the programme matured, it became appropriate to hold an annual conference, attended by all the research groups within the programme plus other interested researchers and practitioners from both industry and academe. In March 1983, as well, a special conference was held at the NCC in Manchester entitled Distributed Computing - A Review for Industry [2], to acquaint senior technical management in U.K. industry with the work of the programme.

1.3.3. Mailshot

From the earliest days of the programme a monthly mailshot was sent to all participants. The mailshot is a collection of papers submitted by the participants themselves and its contents ranged from draft technical papers for comment through to announcements of forthcoming meetings. A particularly valuable feature has been the inclusion of trip reports. It was made a condition of overseas travel that a trip report should be produced for the mailshot. This has proved a very valuable way of keeping the programme abreast of developments overseas.

1.3.4. Annual Report

From the start, the programme has produced an Annual Report, containing an overview of each project within the programme [1]. This has proved a very valuable introduction to the programme for both industry and academe.

1.3.5. Equipment Pool

At an early stage the DCS panel decided to establish an equipment pool from which investigators could borrow. Initially the pool was stocked with magnetic tape decks, VDU's and modems to improve communications and software interchange between research groups. A key decision was to provide Unix (TM) licences for the programme. As time has passed the pool has grown and now includes local area network equipment (Cambridge Ring), X25 connections and high performance single user workstations (Perqs).

6 Distributed computing systems programme

Electronic communications between projects within the programme have steadily improved, through the provision of hardware to link to either the Universities/ Research councils X25 network, JANET, or British Telecoms PSS network. X25 software for the Unix operating system has been produced by the University of York. This is a good example of one of the benefits of coordination: recognizing the need for this software, the DCS panel funded just one site to produce it. Without coordination there would have been a real danger of many sites embarking upon the same project with consequent wastage of effort.

1.3.6. Technical Backup

SERC operates a number of laboratories to provide specialized services and facilities to SERC funded projects. Within the computer science area, SERC's Rutherford Appleton Laboratory (RAL) provides, for example, large mainframe computers and microelectronics design and fabrication facilities.

The DCS programme has been supported by RAL since January 1978. The Academic Coordinator and Technical Secretary are on the staff of RAL and other RAL staff provide software and hardware support for the programme. RAL support has included:

- Support for the Unix operating system used by the majority of DCS investigators.
- Provision of software to couple the Unix troff text formatting software to SERC's III FR80 microfilm recorder.
- Assembly and distribution of software to drive the Cambridge Ring.
- Construction of 6 Cambridge Rings for the DCS equipment pool and procurement from industry of a further 10 6-node ring systems.
- Procurement, distribution and maintenance of the equipment pool.
- Support and operation of the programme's electronic mail facility.

1.4. RESEARCH THEMES

When the DCS programme was first established, the DCS panel categorized the research into five major topic areas, representing a progression from fundamental theory to novel applications. The areas were:

- Theory and Languages: An adequate theoretical basis for Distributed Computing Systems.
- Resource Management: Distribution of control, allocation, scheduling and organization.
- Architecture.

- Operational Attributes: Particularly reliability and performance.
- Design, Implementation and Application. Hardware and software techniques for development and implementation.

As the programme evolved, projects have clustered around emerging ways in which to structure distributed systems which may be claimed reasonably as emerging ground themes:

1. Loosely-coupled distributed systems. Such systems are multicomputer configurations that do not share immediate memory and can be dispersed over wide geographical areas. Research in this area has been concerned with the overall structure of such systems, requirements for operating systems appropriate to this environment and related programming languages.
2. Closely-coupled distributed systems. Typically systems which do share a common memory. Again research has been concerned with architecture, operating systems, programming languages and applications.
3. Non von-Neumann architectures. Research in this area has been concerned with alternative ways to provide high speed numerical computing and with architectures to support the efficient evaluation of declarative languages.

A fourth major theme in DCS has been concerned with theories of parallel computation and with the development of notations and techniques for specifying and verifying such systems.

The work on loosely-coupled systems can be traced back to the work of Wilkes, Needham and others at Cambridge which led to the construction of the Cambridge Distributed Operating System [3]. A key component in this work was the design of the Cambridge Ring local area network, the design study for which was published in 1975. This work commenced prior to DCS, but later Cambridge work was DCS funded. From Cambridge, this approach spread to the University of Kent and other sites, including York, Keele, Oxford, Strathclyde and Newcastle; all of whom have made their particular contribution to knowledge in this area. A particularly important step came early in 1980 when DCS constructed 6 Cambridge Rings, each of four nodes, to the Cambridge Mark 2 design. The demand for this equipment from research groups was considerably in excess of supply and the panel, recognizing the opportunity to foster the take-up of this result by industry, placed a contract with U.K. industry for the construction of further Ring hardware. This was a formative step in establishing the supply of Ring equipment in the U.K. The availability of common hardware for the pursuit of research in this area had a very beneficial effect in drawing research groups together.

Work in the tightly-coupled systems area can be traced back to projects in Evans' group at Loughborough and Aspinall's group at Swansea (now at UMIST). Grimsdale's group at Sussex were subsequently funded to work in this area. Dixon at Hatfield Polytechnic and Evans have explored the application of such systems to various classes of numerical

problems.

The non-von Neumann architectures work can be traced back to projects at Manchester, Newcastle and Westfield College, concerned with the dataflow approach. A seminal event in this area was a Workshop at Newcastle in 1979 which brought together the dataflow researchers and researchers in the fields of applicative and logic languages. This led to a number of proposals for ways to exploit parallelism for efficient execution of such languages and for further language development work. DCS has been largely responsible for the creation of the strong U.K. research community in this field.

Our fundamental understanding of concurrency has been greatly enriched by the work of Milner and Plotkin at Edinburgh and Lauer at Newcastle. It is only proper to acknowledge the great contribution of Hoare's group at Oxford to this field, though this was not funded entirely by DCS. The recent work by Hoare's group on the specification of distributed systems has been funded by DCS and is described in the chapter by Morgan and Gimson. Cunningham, Kramer and Abramsky at Imperial College have also made significant contributions to this area. Design methodologies for distributed systems are discussed in the papers by Hughes and Powell, and Sloman and Kramer.

Performance modelling of distributed systems has been extensively investigated by Mitrani at Newcastle and a review of the area appears later in this book.

1.5. INDIVIDUAL PROJECTS

It is useful to describe a small number of the projects funded by DCS, by way of illustration and to set in context the subsequent chapters in this book. Descriptions of all the projects funded may be found in [1].

The Manchester Dataflow Project has demonstrated the viability of a parallel computing system based on the dataflow model of computation, which exploits irregular parallelism at the instruction level. It allows a wider range of applications than the more rigid vector and array processors. The prototype machine has demonstrated performance improvements through concurrency almost lineal for up to 10 processing elements. This project has delivered concrete results where previously there was only speculation. The prototype hardware is being used both by research institutions and by industry to assess the direction of future dataflow products.

In the declarative architectures field, the ALICE Project is investigating the development of applicative languages, their use in real-world problems, formally based development systems and implementations on highly parallel architectures. This project has received a great deal of public interest and has produced significant papers on language design, programming methodology and computer architecture.

Turner (Kent) has made great contributions to the areas of declarative language design (SASL, KRC, Miranda) and evaluation (combinators). Henderson (Stirling) has explored the problems of producing purely

functional operating systems. Sleep (East Anglia) has also explored the distributed evaluation of applicative languages.

Within the loosely-coupled systems field, the Unix United work of Randell's group [4] has received wide acclaim. This development was funded by DCS and is being exploited commercially. Distributed file-stores and operating systems have also been investigated at Keele (Bennett), York (Ward), and Strathclyde (Shepherd). Bornat and Coulouris (QMC) have investigated one approach to the construction of such systems (Pascal-m)

The main groups in the tightly-coupled systems area are those of Aspinall (UMIST), Evans (Loughborough) and Grimsdale (Sussex). Each has constructed a model system and explored its applicability to a range of problems.

The work of Milner and Plotkin on theoretical models of concurrent systems has received world-wide acclaim. Several U.K. companies are exploring the applications of these techniques to their application areas. Cunningham's group (Imperial College) and Hoare's group (Oxford) have made significant advances in the specification of concurrent systems.

1.6. SOME STATISTICS

The box (Fig 1.1) illustrates the scale and breadth of involvement which DCS has created and managed over the past few years. In particular, we believe that the investment which the programme has made in infrastructure and coordination, amounting to about 27% of the funds expended, has enhanced the overall value of the activity enormously.

1.7. ACHIEVEMENTS

The major achievement of the DCS programme has been to create a strong research community in the U.K.

It is not too strong to claim that without DCS the discussions which lead to the formation of the Alvey programme could not have taken place. DCS has also established new research groups where none previously existed and has enabled a number of young researchers to become established in the field much more rapidly than otherwise would have been the case. The human interface between the management panel and its clients through the coordinators has been a key factor in bringing this about.

The establishment of the U.K.'s strong position in declarative systems research owes much to DCS.

Now that the DCS programme has ended, the work funded by DCS will continue through either the Alvey Directorate or the SERC's Computing Science Sub-committee as appropriate. Research ideas fostered by DCS are appearing in products through the Alvey programme.

Many of the lessons learnt in the DCS programme have been

10 Distributed computing systems programme

Normal research grants awarded	103
Cooperative grants awarded	3
Visiting Fellowships awarded	20
Universities holding DCS grants	23
Polytechnics holding DCS grants	3
Number of research staff employed (approx)	150
Total value of grants awarded	6.3M
Expenditure on coordination	0.4M
Expenditure on infrastructure	2.0M
Total expenditure	8.7M
Fractional spend on infrastructure	23%
Fractional spend on coordination	4.6%

Fig.1.1

incorporated in the Alvey programme, for example the need for full-time technically competent staff to manage the programme and the need for infrastructure both in terms of a workshop programme and computing resources.

This book, plus the research publications of the participants in the programme, mark the intellectual achievement of the programme. In terms of contribution to knowledge, the programme should claim, for example: the advance and earliest use of local networking technology through the Cambridge Ring; the development of new architectural techniques, particularly for Dataflow and Graph Reduction systems; creation of some of the first techniques for specifying and describing concurrent computation; and methods for performance modelling and analysis in complex systems.

Technology transfer to industry is much harder to estimate. The main products of the DCS programme are ideas and demonstrations of ideas, rather than systems that can, and should, be turned directly into commercial products. Trained manpower has also been a major product of the programme, and it is through this avenue that technology transfer is best being achieved. More than one company, has benefitted directly from DCS manpower!

1.8. ACKNOWLEDGEMENTS

The DCS programme has involved a very large number of talented researchers, whose contributions to the programme I wish to acknowledge globally here. I also wish to acknowledge the contributions made by the former chairmen of the DCS panel, Iann Barron, Ian Pyle, and Roger Needham, together with all who have served on the panel; and above all the coordinators of the programme, Bob Hopgood, Gill Ringland, Rob Witty, Jeremy Tucker, David Duce and Fred Chambers.

REFERENCES

1. DCS Annual Report, available from Dr D. A. Duce, Computing Division, Rutherford Appleton Laboratory, Chilton, Didcot, OXON OX11 0QX
2. Distributed Computing - A Review for Industry, Proceedings available from Dr D. A. Duce, address as above.
3. R. M. Needham and A. J. Herbert, The Cambridge Distributed Computing System, Addison-Wesley, 1982
4. D. R. Brownridge, L. F. Marshall and B. Randell, The Newcastle Connection - or UNIXes of the World Unite, Software Practice and Experience, Vol. 12, No. 12, December 1982

Chapter 2

The Sussex broadband LAN project

F. Halsall

2.1 INTRODUCTION

This project evolved from a study conducted some time ago into the provision of a flexible data transmission system for use on the University campus. All the existing data transmission services on the campus at that time had been provided by installing ad hoc wire systems as each requirement arose with the effect that each new requirement necessitated the installation of a new set of cables. The planned expansion in the type and range of computer services to be provided over the next few years meant that it was essential to investigate the provision of a flexible underlying data transmission system which, ideally, could support not only the existing and planned services but also have sufficient flexibility and capacity to allow for future requirements to be met without the major upheavals that are currently involved.

At the time the early investigations were being made, there was much interest being shown into exploiting the use of coaxial cable networks similar to those already in widespread use in the Community Antenna Television (CATV) industry as the basis for providing a wide range of alternative data transmission services on a single cable network. Unfortunately, however, most of the systems being offered at that time were of overseas origin, even though much expertise in this field was available in this country. This project evolved, therefore, as a collaborative venture with a local company - Rediffusion Engineering - aimed at providing the various additional equipments which are necessary to exploit the use of a CATV-based coaxial cable network to meet the planned and future data transmission services of an establishment similar in size to a university campus.

2.2 REQUIREMENTS

Before describing the Sussex network, it is perhaps helpful to first outline the data transmission requirements of an establishment like a university campus. The traditional and still the major service supported at most universities is concerned with providing a distributed community of low bit rate (<9607 bps) terminals accessing one or more

computer systems which are normally located in a centralised building known as the Computer Centre. To add flexibility, many Centres have installed Terminal Switching Exchanges (TSEs) which then allow the terminals to gain access to any of the Centre's machines on switched basis.

As the cost of computing hardware has fallen over the past few years, there has been a trend for some of the larger computer user departments to acquire their own systems can normally meet most local computing needs, however, there are also instances when a user connected to one of these systems also requires access to the more extensive range of services provided on one of the Centre's machines. To meet this type of requirement, either those user terminals which require this facility are connected, not directly to their own local computer, but rather to the Centre's TSE together with a number of terminal ports from the local computer. In this way the user may then login through the TSE either to his own local machine or to one of the Computer Centre's machines. This assumes, of course, that the central TSE has sufficient spare capacity. An alternative solution, certainly if the demand for this type of service is high, is to install a spare satellite TSE in the remote building which is then connected by means of a high bit rate link (typically 1.5 Mbps) to the central TSE. In this way, a user may login either to his own local machine(s) or to one of the Centre's machines via the central TSE. Multiple satellite TSE's may be installed and used in a similar way.

More recently, there has been a trend towards individual users and departments obtaining various personal computers (PCs) to support word processing and other services. Although these are currently being used as single-user stand-alone systems, local user communities with such systems are already planning to be interconnected to share some local resource such as a printer or file system. Moreover, some users are also expressing a wish to use their systems to gain access to wider computing resources from, say, one of the Computer Centre machines. The currently favoured approach to meet this type of requirement is to install a local basedband network-Ethernet, token ring, etc. in each building to allow users in that building to communicate and access local shared resources and then to provide a bridging node into a campus-wide high bit rate (10 Mbps) network. The latter will then also be used to provide a campus-wide facility for computer-to-computer communication.

In summary, the data transmission requirements of a university campus are extremely varied but, it is felt, typical of many similar establishments. When planning a data transmission system to meet the range of services outlined, therefore, it is essential that the selected system has a high degree of flexibility and capacity to ensure that it meets not only the existing and planned requirements but also possible future requirements. The use of a broadband coaxial cable network of the type to be described has this capability.

2.3 BROADBAND DATA NETWORKS

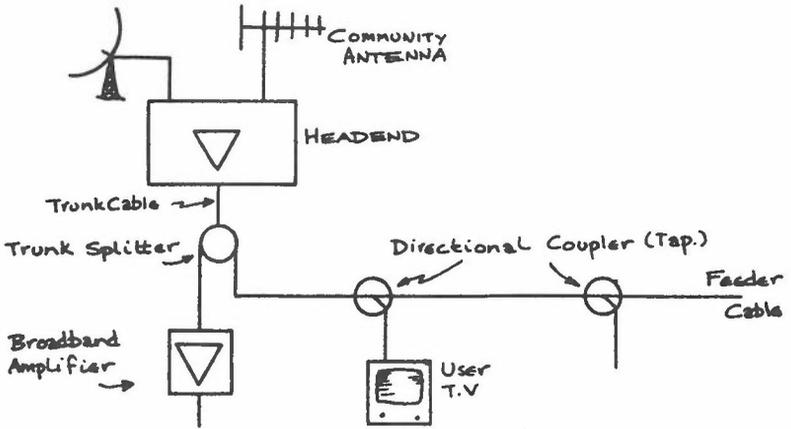
There is currently much interest in exploiting the very wide bandwidth available in a coaxial cable to derive a wide range of data transmission services from a single coaxial cable network. With a baseband cable network-Ethernet, Token Ring, etc. the available bandwidth is used to provide a single, high bit rate channel typically of 10 Mbps. The latter is then time shared between multiple users using a particular media access method. The major use of baseband coaxial cable networks, therefore, is for interconnecting distributed communities of computer-based equipments which can operate and exploit this high bit rate capability.

An alternative way of utilising the high available bandwidth of a coaxial cable, is to use frequency-division-multiplexing (fdm) techniques to divide the total available bandwidth into a number of sub-frequency bands or channels each capable, with the aid of a suitable pair of r.f. modems, of satisfying a particular data transmission service. This approach is known as broadband working and the same principle is currently in widespread use in the CATV industry to multiplex a number of TV channels onto a single coaxial cable. A schematic of a typical CATV system is shown in part (a) of Figure 1. Each TV channel is allocated a particular frequency band, typically of 6 MHz bandwidth, and the received video signals are then used to modulate the selected carrier frequencies. The modulated video signals are then transmitted over the cable network and received at each subscriber outlet.

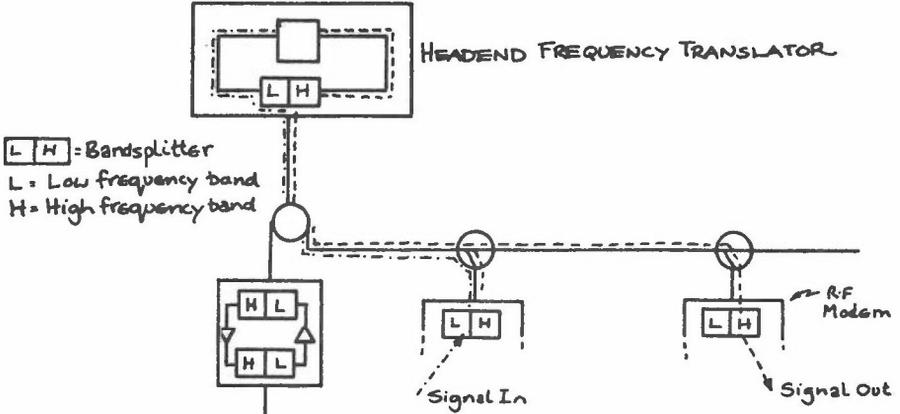
In a similar way it is possible to derive a range of data transmission channels from a single cable by allocating each channel a portion of the total bandwidth; the amount of bandwidth for each channel being determined by the required data rate. For data communication, however, a two-way (duplex) capability is normally required. This may be achieved in one of two ways: either the transmit and receive paths are assigned two different frequency bands on the same cable - single-cable-system - or, two separate cables are used, one for the transmit path and the other for the receive path - dual-cable system. A schematic of each type of system is shown in parts (b) and (c) of Figure 2.1. The main difference between the two systems is that a dual-cable system requires twice the amount of cable and cable taps to install. Nevertheless, with a dual-cable system the total cable bandwidth - typically 5 to 440 MHz - is available in each direction. Moreover, the headend equipment is simply an amplifier whereas with a single-cable system, a frequency translator is required to translate the incoming frequency signals associated with the various receive paths to the corresponding outgoing frequencies used for the transmit paths.

2.4 THE SUSSEX NETWORK

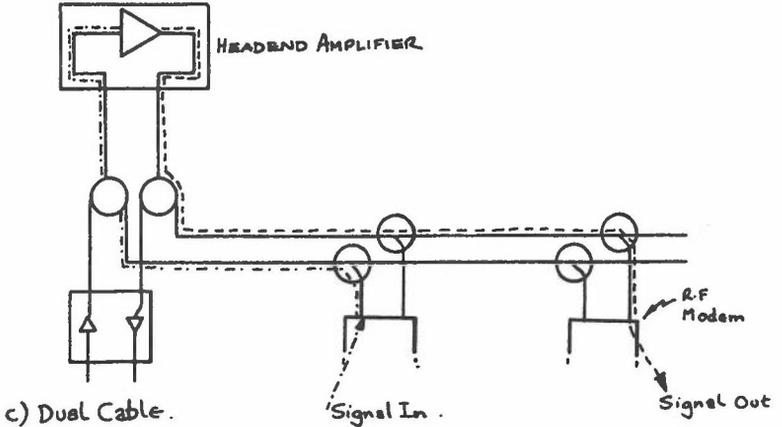
A schematic layout of a cable network suitable for the Sussex Campus is shown in Figure 2.2. Although in principle



a) Basic CATV System components.



b) Single-Cable.



c) Dual Cable.

Fig. 2.1 CATV Network Alternatives

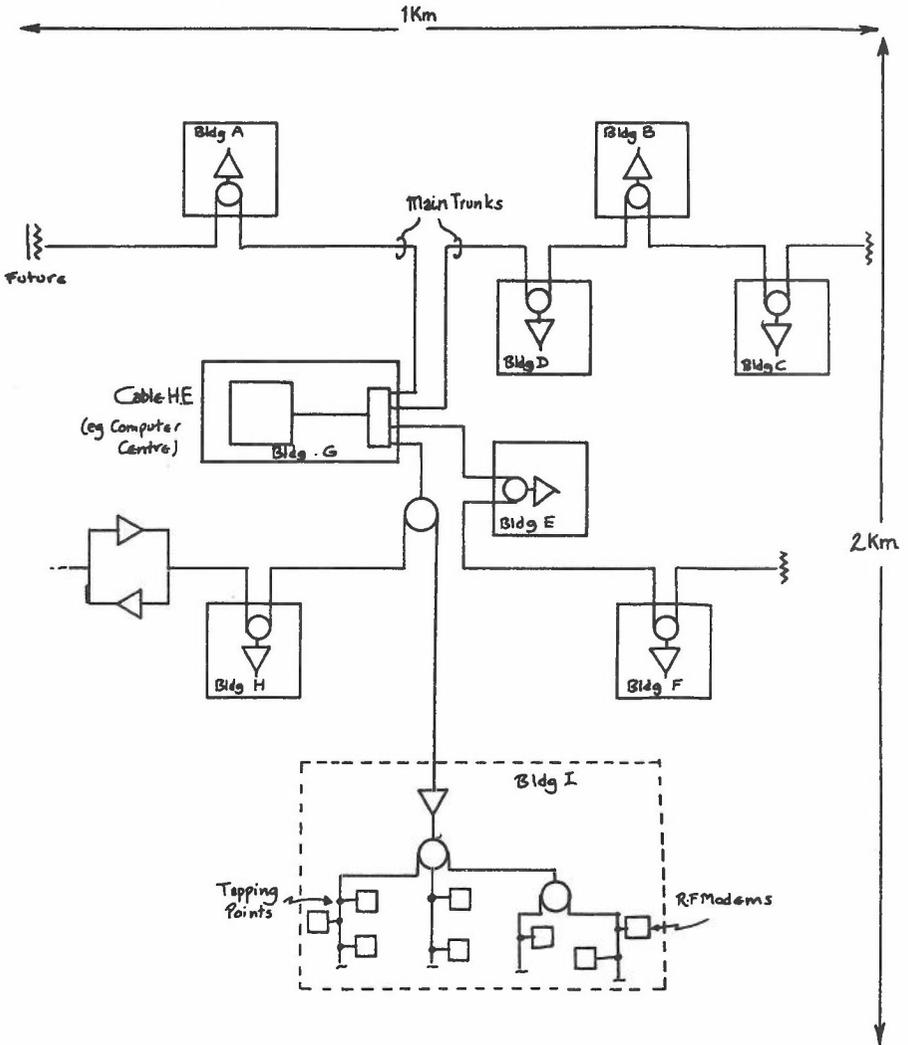


Fig. 2.2 Broadband Distribution Network

a single coaxial cable is used, as can be seen, in the planned system the basic trunk network is a tree topology with the Computer Centre at the root node (headend). The layout of the distribution cable in each building will vary, of course, and hence the aim in the first instance will be to establish a campus-wide trunk network of the type shown with trunk solitters and local distribution amplifiers at the entrance to each building. Sufficient spare signal capacity will then be provided at the entrance to each building to support both the current and projected data transmission requirements in that building. Hence once the trunk cable network has been layed, all future cabling will be constrained to within each building.

Most of the equipment shown in Figure 2.2 is readily available from any of the many manufacturers of CATV components. The major part of the work associated with this project, therefore, has been concerned with the design and production of a range of r.t. modems suitable for use with this type of network. The prototype modems currently being evaluated are:

- a low bit rate (<19.2 Kbps) asynchronous/synchronous duplex modem for use with a set of dedicated point-to-point or multidrop channels;
- a low bit rate (<19.2 Kbps) microprocessor-controlled frequency agile modem for providing a switched communications facility between a community of devices;
- a high bit rate (10 Mbps) CSMA/CD modem to provide a high bit rate channel suitable for computer-to-computer communication.

The design of each type of modem will now be described.

2.4.1 Low Bit Rate Fixed Frequency Modem

For ease of implementation and minimum cost, this modem uses simple FSK modulation. Each 19.2 Kbps full duplex modem requires a total of 80 KHz of bandwidth - 40 KHz for the transmit path and teh same for the receive path. Thus a 6 MHz band - as used for (U.S.) television - can support 75 such modems. To achieve flexibility, the transmit and receive frequencies of each modem are derived from a frequency synthesiser whose frequency is controlled by a suitable binary control word set on a pair of DIL switches within the modem.

A schematic of the r.f. circuitry within this modem is as shown in Figure 2.3. The data to be transmitted are first modulated - part (3) - onto a fixed intermediate carrier frequency and the resulting modulated signal is then band-limited and filtered. This signal is then mixed with the output of the frequency synthesiser which shifts the mixed signal into the desired output frequency band. A bandpass filter is then used to suppress the unwanted mixing products.

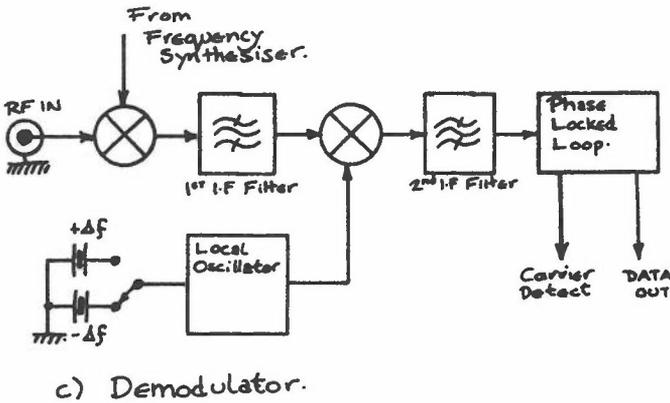
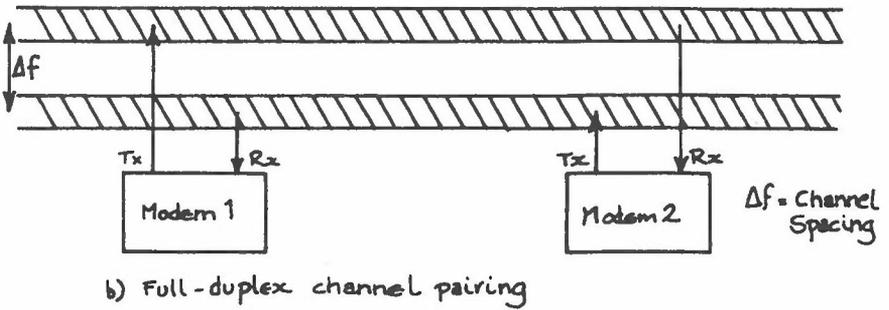
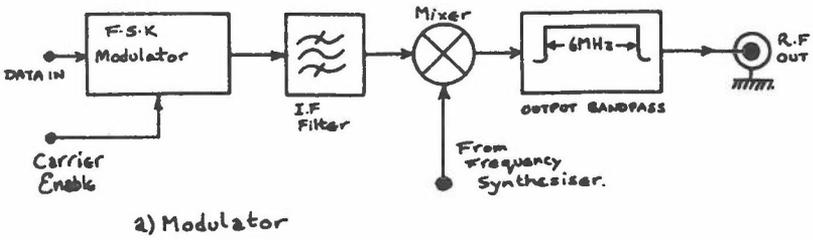


Fig. 2.3 Low Bit Rate Modem Schematic

A reciprocal process is used in the receiver section. The receiver takes the form of a superheterodyne: the incoming signals are mixed with the frequency synthesiser which places the desired signal in an I.F. 'window' after which F.M. demodulation takes place using standard phase-lock-loop techniques. Such a system, tuned by a single frequency synthesiser, would mean that the receiver would also be tuned to the transmitter frequency. For full-duplex operation, however, two separate frequencies must be used as shown in part (b) of the figure. To achieve this, a dual conversion is performed in the demodulator - part (c). The second local oscillator is switched up or down in frequency by one channel spacing by means of a switch within the modem thus providing the necessary shift between the transmit and receive frequencies. The receiver design shown is intended for a dual cable system but a single-cable design is readily achieved by down-converting the receive frequency band to its original transmit frequency before demodulation. A standard RS-232C interface is provided at the user interface to the modem.

2.4.2 Frequency Agile Modem

This design of modem is intended to provide users with a low bit rate (<19.2 Kbps) switched communication facility between a community of devices which support a basic RS-232C interface port; in this instance the cable is being used to provide a form of distributed switch.

The r.f. section of this modem is very similar to that just described except that the transmit (and hence receive) frequency is controlled by a microprocessor rather than simple switches. The control protocol of the modems is similar to that used for CM radio: there is a common signalling channel - operated using CSMA/CA techniques - to which all other modems are tuned when not in use. Each modem has a unique network-wide address and, whenever a user wishes to communicate with another user, the user enters the required destination address. The controlling microprocessor (within the modem) first scans the available channels to find a free channel. It then creates a frame with the required destination modem address and the selected channel number within it and broadcasts this on the common signalling channel when the latter is free (quiet). It then controls the r.f. circuitry to move to the selected channel to wait for a response.

Assuming the called modem is not busy, it receives the calling frame on the common channel by detecting its own address in the frame header and then causes the r.f. circuitry to move to the selected channel. It then responds with an acknowledgment frame on the selected channel. Both microprocessors then connect their RS-232C ports to the input of the r.f. section and communication between the two user devices can then commence, the presence of the r.f. modems being transparent to both users. Either user can close the connection at any time simply by pressing a key on the modem. Also, there is a timeout mechanism operated

by the microprocessor: if there are no transmissions on either channel for a set period of time, the connection is automatically cleared and the user informed.

Contention on the common signalling channel is resolved using a technique known as Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). When in the calling state, both the transmit and receive sections of the modem are tuned to the signalling channel. The receiver is used therefore to provide the carrier sense signal. When the originating modem has found a free channel and wishes to send a connection request frame on the signalling channel, it first waits until the carrier sense signal becomes false. It then waits a further short random time interval and, if the channel is still free, sends the frame. In addition, the calling modem remains on the signalling channel until it receives the connection request frame back via the cable headend. Then, if this is corrupted, it will repeat the calling procedure again. If it is correct, however, it moves to the previously selected channel to await the receipt of an acknowledgment frame from the called modem.

If a correct acknowledgment is received the procedure is as above. If no acknowledgment is received within a set time interval, the called modem is assumed to be either busy or disconnected. The calling procedure is therefore repeated a second time but if the called modem does not respond to the second connection request a 'destination-not-available' response is fed back to the user.

2.4.3 High Bit Rate Modem

The two previous types of modem are intended for low bit rate applications. In addition, however, it is the aim to produce a range of high bit rate modems (<1 Mbps) suitable for both dedicated point-to-point applications and also switched applications. With respect to the latter a prototype modem design is now operational which supports a 10 Mbps CSMA/CD (Ethernet) channel on the cable and its design will now be outlined.

The aim of this design is for the modem to present a transparent interface to the host system; in practice, the user interface is the same as that used for a transceiver unit connected to a baseband cable. An important aspect of the modem design, therefore, is to perform the collision detect function. With a baseband system this is readily performed since simple electronics within the transceiver detects if the signal present on the cable is different from that being transmitted and, if it is, the collision-detect control line is activated and the host ceases transmission. In the broadband system, however, the transmit and receive channels are different. Moreover, data transmitted on the transmit channel are not received until the signal has propagated through to the headend of the cable and back again; collision detect by instant comparison of the transmitted and received signal is therefore not possible.

One obvious solution is to retain a copy of a certain number of bits transmitted in, say, a shift register and to

compare the contents of the shift register with the bit stream being received. Problems arise with this method, however, since the received signal levels of the two (or more) colliding modems (stations) may differ by as much as 6 dB - a feature of the cable design. This means that the modem with the higher signal level still receives the data correctly as it dominates the channel and hence only the modem with the lower signal level can detect the collision.

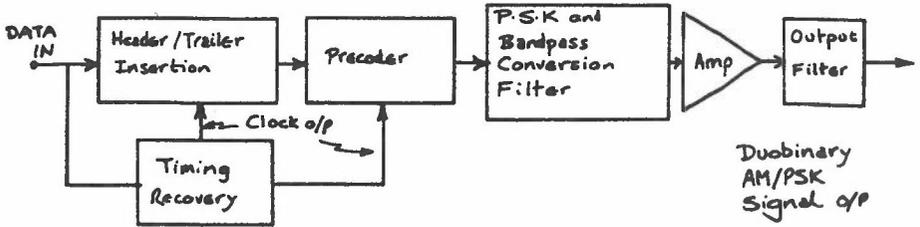
The solution adopted, therefore, endeavours to overcome this problem by exploiting the modulation technique being used. High bit rate channels such as Ethernet - which employs Manchester encoding - require a significant portion of the cable bandwidth. To minimise the amount of bandwidth utilised, therefore, duobinary AM-PSK modulation is used. Normally, with this method only the amplitude of the demodulated signal is recovered to derive the baseband data. By also examining the phase variations of the received signal, however, a reliable method for determining non valid - and hence corrupted - duobinary AM-PSK signals can be obtained.

A block schematic of the CSMA/CD modem is shown in Figure 2.4 and Figure 2.5 illustrates a typical set of waveforms produced by the modulation and demodulation circuitry. The baseband data are first passed into the precoder. The function of the latter is to count the number of 1's in the incoming data stream; when there is an odd number, the output is high (1) else it is low (0). This signal is then passed to the PSK generator which produces a bi-phase PSK signal corresponding to the states of the precoder output. Finally, the PSK signal is fed to a bandpass filter. The filter is an important element in the modulator since it performs the function of adding the PSK signal of the previous bit to that of the current bit. This has the effect of producing an AM signal whose amplitude corresponds to the baseband data. Another function of the filter is to limit the bandwidth to a minimum; for example, a 10 Mbps channel requires only 5 MHz (3dB) of bandwidth.

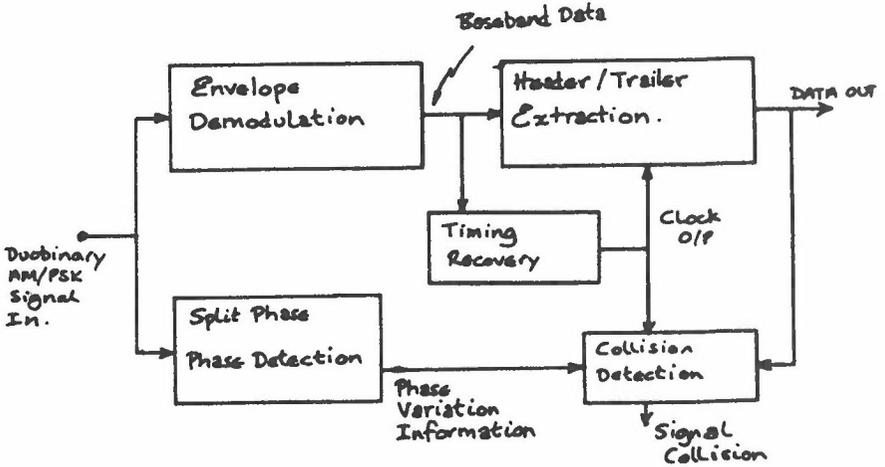
The recovery of the baseband data from a duobinary AM-PSK signal is straightforward and hence needs no further explanation. As has been mentioned, however, it is also possible to use the phase variations associated with this type of signal as a means for detecting a collision. Careful examination of the waveforms shown in Figure 2.5 will show that the phase variations of the encoded waveform are related to the data stream: when there is an odd number of 1's between two strings of 0's, there is a 180 degree phase shift in the carrier representing the two strings of 0's; if there is an even number of 1's there is no phase shift in the carrier. Hence, by recovering the phase variation information from the received signal and comparing this with the recovered baseband data, it is possible to detect when a corrupted AM-PSK signal has been received. Since the noise level of a broadband cable is very low, this can reliably be interpreted as an occurrence of a collision on the cable.

2.5 CURRENT STATUS AND DISCUSSION

At the time of writing this paper (April 84), a number



a) Modulation



b) Demodulation.

Fig. 2.4 High Bit Rate Modem Schematic

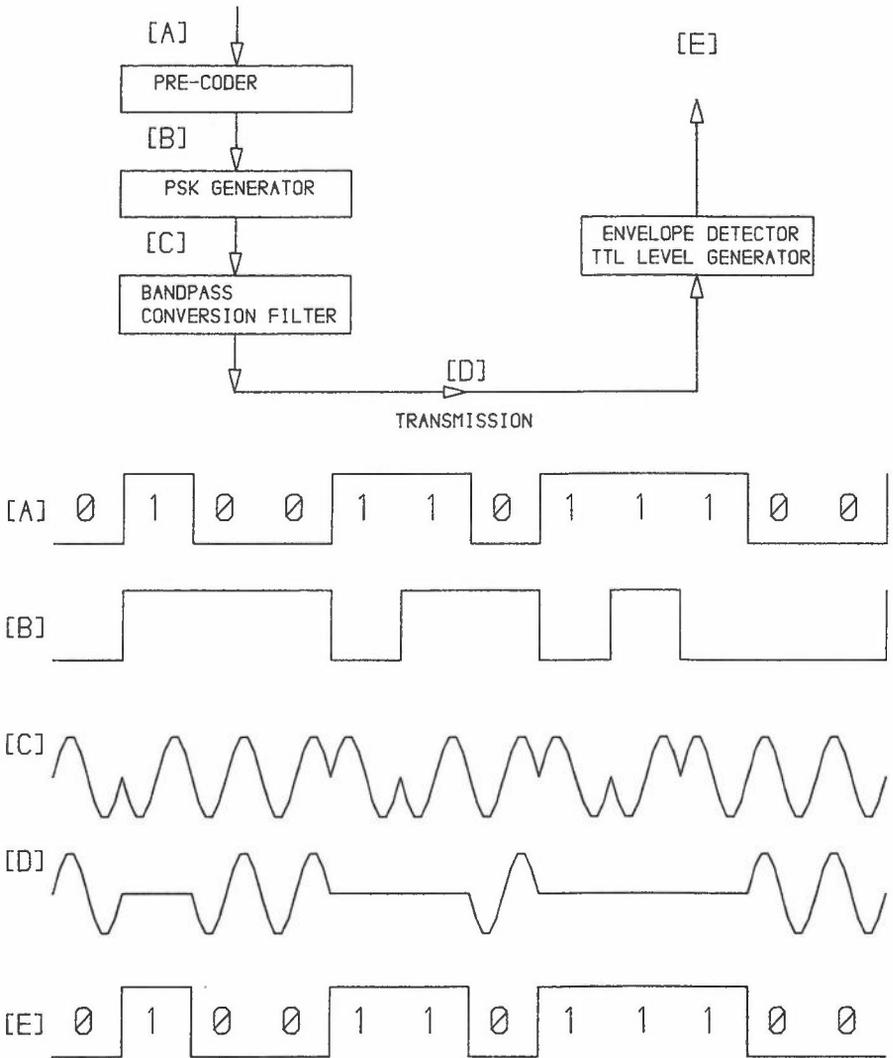


Fig. 2.5 Duobinary AM-PSK Principles

24 Sussex broadband LAN project

of modems have been built based on the outlined designs and are currently being tested on a laboratory experimental cable system. When these tests have been completed, it is planned to test the modems on a skeleton trunk cable network which effectively links two buildings to the Computer Centre. All indications to date are that the modems will operate satisfactorily and coexist on a more substantial network. Since most of the trunk and feeder cable components are known to be very robust, it is strongly felt that a broadband coaxial cable distribution network of the type outlined offers a viable alternative for meeting the wide and diverse data transmission requirements of an establishment like a university campus.

Implementation of a high performance LAN —Centrenet

D. A. Edwards, R. N. Ibbett, T. P. Hopkins

3.1 INTRODUCTION.

Historically, there have been two main motivations for the interconnection of processors to form a network. Firstly, users on different machines, often on different sites remote from one another, have had the need to exchange data on an irregular basis. This has led to the development of wide-area-networks (WANs) of varying complexity, typically using telecommunications technology for the interconnections. Such links have tended to be slow and used primarily for file transfer operations; latterly electronic mail and "bulletin board" activities have become increasingly important.

The second motivation for networking has been that users, local to one site, have wanted to be connected together in order to increase the facilities or computing power available to all. The local interconnection may be that of a number of processors tightly-coupled in order to form a high speed multi-computer system. An example is the MU5 computer system (Morris and Ibbett (1)) designed and built in the Department of Computer Science. In other cases, the local interconnection is a loose coupling of a number of processors to provide a sharing of expensive resources (fast line printers, archiving facilities, number-crunching CPUs etc). This form of network is generally referred to as a local-area-network (LAN), and has become increasingly important as the cost of processing power has decreased, and computing resources have become more decentralised. LANs have generally traded the speed of tightly-coupled multicomputer systems for increased size and flexibility. The differences in use between WANs and LANs have become blurred. Wide-area networks support resource sharing either of hardware or of software packages; for example SERC's Rutherford Appleton Laboratory provides VLSI design facilities for remote users. On the other hand, LANs are used for file transfer between disparate machines and operating systems and for network mail. An aspect of networking which should not be overlooked is the extent to which a user community is fostered.

26 High performance LAN-Centrenet

The Department of Computer Science has a long history of the design and implementation of hardware and software for large computer systems (Lavington (2)). The successor to the MU5 system is a hierarchical multicomputer complex known as MU6 (Edwards et al (3)). The MU6 proposal required high speed interconnection of processors, but with more flexibility than that provided by the MU5 switching scheme, the MU5 Exchange. An investigation into the hardware required, and an examination of existing LAN technology (Hopkins (4)), resulted in a realisation that a more general approach should be adopted.

3.2 CENTRENET PHILOSOPHY.

The MU6 project is but one of a number of many research activities in progress in the Department. Each research group typically makes heavy demands on its own set of resources but occasionally needs to attach itself to other computers or peripherals. The situation existing within the Department is a reflection of that existing on the campus generally. Here there are many groups of users who are concentrated in a number of buildings, which are scattered over several kilometers. Most have computer facilities of their own but need to communicate with UMRCC (the University of Manchester Regional Computer Centre). A network capable of catering for all these users would have to provide not only a large number of machine connections, but also a very large number of terminal connections. The environment described above is also true of many large industrial sites which have computing units in different buildings.

In addition, therefore, to meeting the needs of the MU6 complex, it was decided that any network chosen should reflect the actual topology of usage: localised clusters of users requiring communication with other clusters. Higher transfer rates are required for local traffic (i.e. within a cluster) than for remote communication (intercluster communication). Thus both high connectivity, in order to accommodate the potentially large number of terminal connections, and high localised bandwidth is required. It was felt that existing LANS such as the Cambridge Ring (Wilkes and Wheeler (5)) and Ethernet (Metcalfe and Boggs (6)) did not reflect the real topology of usage described above. They have the disadvantage that, for various practical reasons, large numbers of connections may only be obtained by linking together subrings, or subnets by means of gateways or network bridges. A further drawback of both these systems is that bandwidth is shared among all users. This may be of little consequence if the main purpose of the network is file transfer, even if the files involved are large, but the origins of Centrenet were in the tightly-coupled multicomputer system where bandwidth sharing could be a severe disadvantage. Indeed, one of the purposes of the network was to act as an enabling technology to allow other groups to investigate fully-distributed computing including such aspects as demand-paging across the network. Concurrent with these investigations into possible network

architectures, interest was being shown in the Department in the emerging technology of fibre-optics. Data transmission by optical fibre has a number of advantages. In the context of civilian computer traffic, the most important are the absence of ground-loops, electrical isolation, the absence of induced noise on long cable runs and for links between building, the freedom from lightning-strikes. Conventional copper wire systems across the campus have suffered from all of these problems. An additional benefit of optical fibre transmission is a greatly superior bandwidth-distance product compared with co-axial cable systems. High speed communications can be maintained between sites several km apart without the need for repeater stations. Cambridge rings have been implemented with optical fibre and an interesting version of Ethernet known as Fibernet (Rawson and Metcalfe (7)) has been built, but neither of these networks is especially suited to fibre-optic technology, which is best adapted to point-to-point links. For these reasons it was decided to investigate the design of a new LAN incorporating optical-fibre technology which would enable the Department to gain some expertise in this area.

3.3 CENTRENET ARCHITECTURE.

Centrenet is organised as a tree-structured hierarchy of high speed packet switches as shown in Fig. 3.1. Each node of the tree is known as a Starpoint. Connections between Starpoints are serial and based on optical fibre links; processors may be attached either directly to a Starpoint or by a serial link. Until recently star-type networks have not been very popular, presumably because of fears that should the central node fail, the whole network ceases to function. Interest in this topology of network has increased and a number have been described in the literature (Sikora and Franke (8), Lee and Boulton (9)). The Centrenet architecture has the following properties:

1. It is modular.
2. It supports "computing clusters". Clusters may be several km apart.
3. It has high localised bandwidth.
4. It has high potential connectivity.
5. It is suitable for implementation in fibre optic technology.
6. It should have high reliability. Furthermore, should a Starpoint fail, the effect is to partition the network into two parts. Only communications which pass through that Starpoint are affected.
7. Each node incorporates "intelligence".
8. There is only one route between any two connections.
9. The network is asynchronous; different links in the network can operate at different speeds.
10. Handshaking of each packet occurs at each stage of its transfer across the network. There is an optional facility whereby a packet may cause an end-to-end acknowledge and/or echo of the packet. This facility is built into the hardware and requires no overhead on the

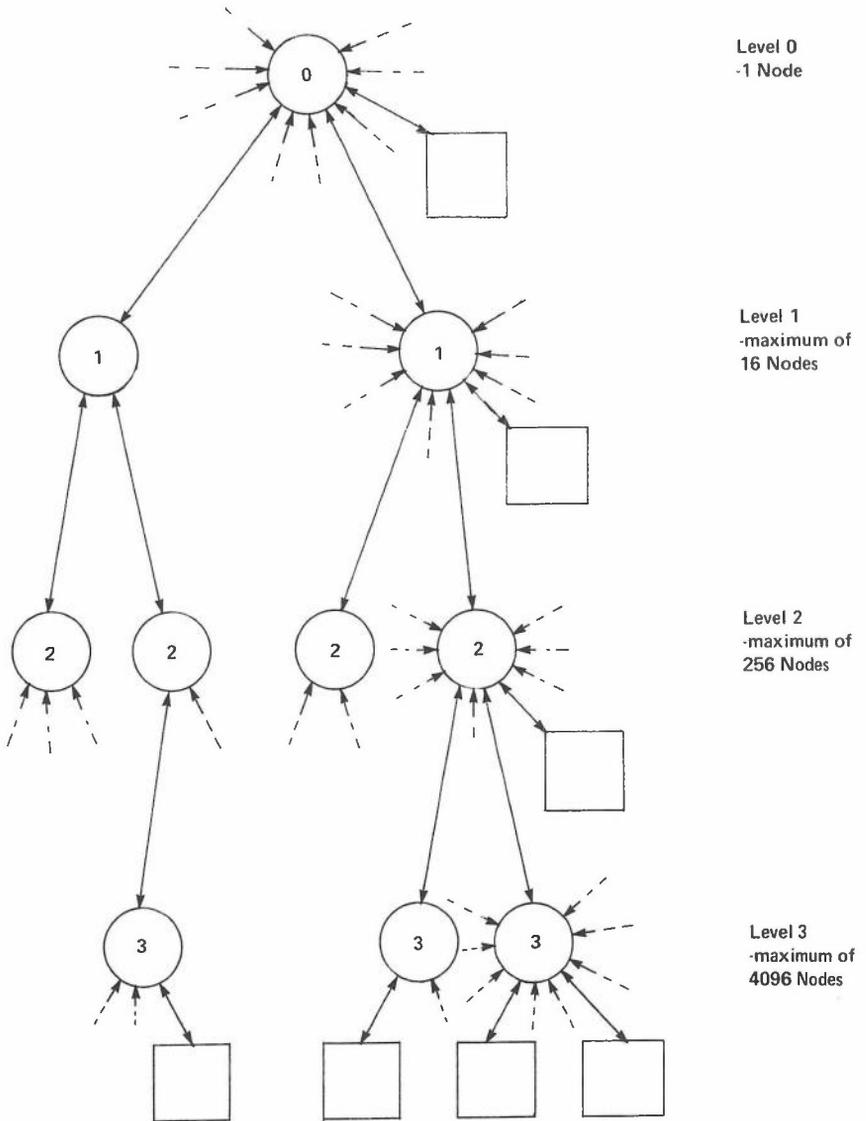


Fig 3.1 The Centretnet Hierarchy.

part of the attached device.

In order to achieve high bandwidth at the nodes, parallel switching of packets is required. This is achieved by a conventional bus design which is easy to implement. The bus may be designed so that in the event of a particular port on the bus failing, normal operation of the rest of the system is unaffected. The reliability of such a system is high, being similar to that used in many computer systems.

The choice of a parallel switching node influences, because of engineering constraints, much of the rest of the network architecture. The Starpoints are built around 4-layer treble Eurocard (9U) PCBs. This large board size has become a Departmental standard allowing high-complexity systems to be partitioned on to one PCB, and gives a reasonable number of connector pins (3 sets of 64 signal pins). Nevertheless, if parallel switching is employed the packet size is limited. Increasing the packet size increases the cost of a system by increasing the number of connector pins required and the number of integrated circuits required in the packet registers. A packet must contain addressing information to identify both the source of the packet and its destination, a number of control bits associated with flow control and error checking, and the user data itself. These fields are conveniently manipulated in multiples of 8 bits. An 8 bit address field limits the number of connections to 256 which is too small for with the environment envisaged. A 16 bit address allows 65536 connections which is probably adequate. It was felt that at a Starpoint no error checking bits were required and thus the control field is only 8 bits - concerned mainly with handshaking of packets across the network. A Centrenet packet therefore has an "overhead" of 40 bits. A large data field increases the efficiency of block data transmissions, but is inefficient for single character traffic such as that generated by terminals. However, in the context of the high performance anticipated from Centrenet, such inefficiencies are irrelevant. Thus it is desirable to make the data field as large as is possible. If the data content were to be 16 bits, the packet utilisation efficiency would be only 28.5%, whereas a 32 bit data field would increase this figure to 44.4%. A 64 bit data field would be even better, but the cost of implementation would then become too high. The Centrenet packet size is therefore a compromise, consisting of a 16 bit source address, a 16 bit destination address, a 32 bit data field, and an 8 bit control field.

A Starpoint consists of 15 ports, 1 uplink port, a Network Intelligence Module (NIM) and a microprogrammed bus controller card which also acts as a NIM interface card. The NIM is a single board microprocessor concerned with set-up procedures, fault-reporting and local name-serving. It does not stand directly in the path of any packet switching. A Starpoint may be used in a stand-alone configuration, or may be connected to another by linking the uplink port to the port of another Starpoint. In a fully

connected system, a 4-level tree is constructed allowing 50,625 connections at the lowest level. In a partially populated system, the tree need not be symmetric. The link between two Starpoints is known as a "remote link", since it is anticipated that Starpoints could be spread across the campus. The Starpoint concept and implementation has changed since the start of the project and it is interesting to note its evolution.

3.4 MARK-1 STARPOINT DESIGN.

Connection between processors and the Starpoint is via a "port card". A port-card is essentially a set of registers under microprogram control. The port-card can receive a packet either from the Starpoint bus or from the device attached to the port-card. Two interfaces are provided on the mark-1 port-card: a 72 bit parallel packet port and an RS232 serial port. If the device attached to the parallel port wishes to send a packet to another port on the Starpoint, it must first load the packet to be transmitted into its own transmit packet-buffer register. The transmitting device must then assert a "Tx Request" bus control line. This instructs the Starpoint interface of the port-card to place the packet on the Starpoint bus the next time the port is polled by the bus controller. One of the other ports in the Starpoint will recognise the packet destination address and accept the packet. This port will send a "Frame Ack" signal back along the bus a fixed number of clock periods later to indicate the receipt of the packet. The transmitting port-card will then assert a "Packet Sent" signal to the attached device, which must then de-assert the "Tx Request". The port-card then deasserts "Packet Sent" and the port is then ready to send another packet.

When a packet is received from the internal bus by a port-card, it asserts an "Incoming Packet" signal to the attached device. The device interface must assert a "Busy" interface signal and the port-card will write the received packet into the device's receive packet-buffer register. The device must then remove the packet before de-asserting "Busy". The incoming packet handshake sequence is then completed.

The essential feature of this mechanism is that each packet transfer across the Starpoint is explicitly acknowledged, and this principle is extended to each stage of a packet's transfer across the network. If no acknowledgement is received, the transmitting port-card retries at the next opportunity. Eventually, if the transmission can not be carried out the port-card times-out, reports the error to the NIM and discards the packet.

If the attached device uses the serial interface, the mechanism is essentially the same, however only one byte of the data field is used.

The rate at which packets may be routed between any pair of ports will depend on the speed at which the receiving device can react to the "Incoming Packet" signal. Also, transmissions will be slowed down if more than one port is sending to the same Starpoint port. However transmissions which do not have either source or destination addresses in common will not degrade one another's performance. The mark-1 port-card is not able to support simultaneous transmit and receive to/from the same pair of ports so that a maximum of eight transfers may take place in parallel without affecting the system's performance. The time taken for an individual transfer depends on precisely when "Tx Request" is asserted. Under the most favourable circumstances 20 clock cycles are required (=3.2 microseconds), however if the request just misses the previous bus slot, the port-card must wait until it is polled again. At present polling is carried out on a round-robin basis so that a further 17 clocks are required before the device is able to transmit. It should be noted that the actual polling sequence is table driven and can be dynamically changed by the NIM allowing the possibility of adaptive polling. The average transmission rate is therefore 15.8 Mbits/sec giving a maximum throughput across a Starpoint of 126.4 Mbits/sec.

3.5 MARK-2 PORT-CARD

The mark-1 port-card has a number of disadvantages. Much of the logical complexity of the port card is due to the fact that there is both a serial interface and a parallel interface on the same card. Whilst direct terminal connection on to Centrenet is considered important, dedicating a port-card to a single serial channel is an extremely expensive method of achieving this goal. In order to reduce the cost of terminal connections, two terminal-multiplexing designs have been investigated. One is a 16 channel bit-slice processor design and the other is a cheaper, more flexible, microprocessor design supporting 8 channel operation but with inevitable performance limitations. It seems likely that the latter design will be adopted. Since a separate terminal-multiplexor was available, it is sensible to remove the serial channel from the port-card. The board has been redesigned taking advantage of more recent semiconductor technology; this has reduced the chip-count for the packet registers by a factor of four. The control logic has been redesigned using PAL technology, which has allowed both a further reduction in the chip-count and the number of clock cycles required for the packet switching operations. The Starpoint clock at present runs at 5 MHz. Only 4 clock cycles are required for a transmit sequence and 6 cycles for a receive sequence. Since these operations are performed in parallel, the transmit time is dominant, and thus under best-case conditions, a packet may be transferred in 1.2 microseconds, giving a maximum bit-rate through one port of a Starpoint of 60 Mbits/sec. The calculation for the maximum throughput across the Starpoint with all channels active is slightly

different from that of the mark-1 Starpoint. Firstly it is now possible to support simultaneous duplex transmission between a pair of ports, and secondly the round-robin polling time ($=17 \times 200$ nsec) is greater than the time taken to switch a packet. Thus, under the most favourable conditions, 8 sets of two-way transfers could be occurring, giving an overall capacity of 338 Mbits/sec.

The reduction in the number of chips required for the packet switching function has meant that board area has been freed for other uses. The new port-card logic has been laid out in an extremely compact manner on a 4-layer printed circuit board; it consumes only about 10% of the board area. The rest of the board is a general purpose wire-wrap board so that a variety of processor interfaces can be accommodated on the same design of port-card. Some interfaces need to be replicated several times. In these cases a printed circuit board is designed in which the port-logic layout may be incorporated without change.

3.6 THE SUPERPORT.

In order to achieve high speed data transfer from a processor to a Starpoint, hardware is needed to interface the processor bus to the Starpoint port-card. So far one such design, known as a Superport (Hopkins (10)), is operational and provides an interface to the DEC PDP11 Unibus. A Superport provides a multiplicity (16) of network addresses to a processor via a single network connection. The multiple channels may be used to support either terminal-to-processor connections or processor-to-processor connections. A Superport has three modes of operation:

1. Block Mode, is used to transfer large blocks of data (i.e. files) at high speed. A header packet is sent giving details of the transmission to follow. On receipt of an acknowledgement from the receiving device, the transmitter transfers the data in "short blocks" (4 bytes packed into one packet). When the packets are received, they are autonomously transferred into a buffer area using DMA techniques. The final packet of the transfer contains a Cyclic Redundancy Check (CRC) to ensure the integrity of the transmitted data. Once the transfer is complete, the processor is interrupted and if the data is correct, an acknowledgement packet is set to the originating processor.
2. Character Mode is used for terminal traffic that requires single character transmission. When a character is received, it is placed in a buffer in memory and the processor is interrupted. Only when the character has been processed is the acknowledgement packet sent.
3. Line Mode is a variation of Character Mode; transmission is similar to Character Mode, however the receiving Superport only interrupts the processor when a predefined end-of-line character occurs.

The observed bit rate from a Superport is 3.0 Mbits/sec. Whilst this is much less than the available channel capacity of the Starpoint, it should be pointed out that the Superport hardware and microcode were designed for ease of commissioning and a primary objective was to gain experience of such a device in an operating system environment. Several conclusions can already be drawn. The device needs to be more programmable than it is; the microcode should be both wider and deeper than at present. Secondly, it has proved difficult to take advantage of the multiple channel block-mode facility. The problem is in the complexity required in the operating system software of the host machine. On the other hand, the ability to support both a single-channel block-mode transfer and multiple-channel single-character transfers has proved valuable. These points are discussed in more detail elsewhere (10).

3.7 REMOTE LINKS.

While a Starpoint may be operated as a stand-alone network, more generally it will be connected to other Starpoints. The distances between Starpoints within the same building may not be large, but connecting different buildings on the campus may involve distances of several km because of the lack of suitable direct ducts. The Remote Link (Train (11)) has therefore been designed with the following objectives.

1. The link should be an integral part of Centrenet and should follow its design philosophy; in particular each packet should be explicitly acknowledged.
2. The link should be a high speed serial link which should not slow down the operation of a Starpoint.
3. The link should be capable of operating over distances up to 1.5 Km.
4. The link should be based on optical fibre technology using readily available components.
5. The link should incorporate error-checking and recovery.

The remote link was designed concurrently with the mark-1 port-card and was designed to be able to deal with the situation of one port requesting transmission every cycle when polled on a round-robin basis, i.e. every 2.72 microseconds (17 x 160 nsec). In practice the link runs at 40 Mbit/sec, with the Centrenet packet enveloped into a remote link super-packet (RLSP) of 112 bits, and a request can be serviced every 2.8 microseconds. The time of transmission over 1.5Km of optical fibre is 7.2 microseconds. If transmission of the next packet had to await the return of an acknowledgement packet, (i.e. a total delay of 14.4 microseconds) the system performance would be grossly degraded. However, in order to be consistent with Centrenet philosophy, each packet has to be acknowledged at each stage of its transfer through the network. The solution is to provide buffering at both receiver and transmitter. At the receiver buffering is required because a Starpoint port can only hold one packet at a time. If no

buffering were included, any delay in placing the received packet on to the Starpoint bus would result in lost packets. At the transmitter, a buffer is required to store packets that have been sent, but which have not yet been acknowledged. The Remote Link super-packet contains a checksum; if an RLSP is found to be corrupted, an error RLSP is returned to the transmitter and a retry is initiated. The transmitter has an 8-deep buffer, allowing up to 8 packets to be sent over the remote link before awaiting an acknowledgement. The link is thus kept full, with no penalty imposed by the time-of-flight of the packet.

The protocol used by the remote link is based fairly closely on the HDLC procedures [ISO76]. The remote link super-packet contains the following parts:

1. A flag, which is a unique sequence (01111110) of bits indicating the start of a packet frame. In order that this bit pattern cannot occur anywhere other than in the true flag, an extra 0 has to be inserted during transmission after five consecutive 1's have occurred. This process is known as bit stuffing. At the receiver a complementary operation, unstuffing, has to be performed.
2. A frame-header, which contains address and control information. The address field is actually irrelevant because the remote link operates on a point to point basis, and therefore there can only be one address for the super-packet. The control field identifies the type of packet that is being sent, i.e whether it contains genuine data or is an acknowledgement packet, and provides a labeling scheme to number each packet. A numbering scheme is required in order to be able to acknowledge or to reject individual frames.
3. The data field containing the Centrenet packet.
4. A CRC for error checking.

The packet buffers required by the remote-link operate on a first-in, first-out (FIFO) principle. An eight-deep FIFO is used for both transmit and receive buffers, and is constructed from TTL RAMs with a controller built from discrete MSI parts to simulate a true FIFO.

The use of optical fibre for the transmission medium has several implications for the system. One of the advantages of using optical fibre is noise immunity. However while problems of electrical interference are eliminated along the length of the cable run, the problem is exacerbated at the receiver. The signal current from the receiving diode is very small (of the order of microamps). Early attempts to mount the receiver circuitry on the PCB proved rather susceptible to noise generated by the large switching currents generated by the TTL logic. It became necessary therefore to fabricate a receiver module in its own shielded metal box; this solved the immediate problem but caused a packaging problem in that the modules were too large to be directly mounted on the PCB. In order for the superpacket to be sent serially down the fibre, it has to be

encoded to include sufficient timing information to regenerate the clock at the far end. Furthermore, the average DC level of the encoded signal should vary as little as possible. This restriction arises because the small size of the received current constrains the receiver amplifier to be AC-coupled. Any shift in the average DC level of the signal effectively reduces the noise immunity of the level discriminator at the receiver. An obvious coding technique to employ is Phase Encoding (PE) or Frequency Modulation (FM); both schemes are relatively easy to implement. Unfortunately, both methods are expensive in their use of bandwidth and a 40 Mbit/sec transmission rate would not have been obtainable with the optical components (Honeywell "Sweet-Spot" LEDs) which were used. A variety of coding schemes were investigated in detail (MFM, 2/3 codes), but the simplest scheme is to encode a '0' by a transition (NRZI-S coding). The bit-stuffing, which limits the number of '1's ensures that sufficient timing information is included and limits the amount of DC-wander to an acceptable level.

The NRZI-S scheme requires minimal logic to implement. Effectively, the complexity of coding has moved to the bit-stuffing system. Overall, about 200 integrated circuits on two treble Eurocards, were required to implement a transmitter-receiver station of a remote link.

3.8 LOCAL LINKS.

Processors may be physically attached to the Starpoint via an appropriate interface such as a Superport. This arrangement, while giving maximum performance, is rather inelegant, involving the extension of the processor's bus to the Starpoint. In a tightly-coupled computer environment this may be acceptable, but in general machines will be further apart than the maximum distance that a processor's bus can be extended. The local link is a point-to point serial link which allows processors to be placed some distance from the Starpoint to which it is connected. The links have been implemented in both optical fibre and co-axial cable in order to compare the two technologies. Like the port-card, the local link has been redesigned. The experience gained in the implementation of the remote link suggested that a much simpler, more cost effective scheme was required. The philosophy behind the local link was that it should be invisible to any of the existing hardware or software. In other words, it should be possible to connect a Superport either directly to a Starpoint, or indirectly through a local link, without any other changes to the network.

In order to simplify the link, the buffering and packet labelling used in the local link is omitted. In the first version, however bit-stuffing, the NRZI-S coding scheme and a primitive form of automatic error recovery were retained. A second version of the link is even more simplified; the need for flags has been eliminated and consequently so has the need for bit stuffing. The coding

36 High performance LAN-Centrenet

scheme used on the new local links is Phase Encoding. The handshaking sequence for the port-card described earlier is simulated by the link; further packets may not be sent until the acknowledgement has been returned from the far end. Consequently, the serialisation time of both packet and acknowledgement causes an extra delay. At the link rate of 15 Mbits, this adds an extra 8.9 microseconds to the minimum time between successive packets. The cable delay over the distances likely to be covered by a local link is comparatively insignificant. The serialisation delay is inherent in any scheme without buffering which retains the hop-to-hop flow control of Centrenet.

3.9 CONCLUSIONS.

At the present time hardware for a pilot Centrenet system has been implemented and connects 2 PDP11s to a single Starpoint via two local links. A VAX running the MUSS operating system is to be attached shortly and will give experience in realistic traffic rates. The reliability of the present network appears to give an error rate of better than 1 error in $2 * 10^{10}$. The local links are switchable to either co-axial cable or optical fibre. No significant difficulties have been found with the optical system, its reliability appears to be at least as good as that of the co-axial cable. A prototype remote link has been demonstrated over a distance of 1.2 Km. The aim of the Centrenet project was to achieve a high performance LAN. This was to be achieved not only by raw hardware speed and parallel switching techniques, but also by putting as much functionality as possible into the hardware. In one respect, this latter approach has not proved cost-effective. An attempt was made to provide extensive error-checking. In particular attempts were made to guarantee, in hardware, a packet's arrival. This has proved difficult to engineer successfully. Reducing the specification to provide a reliable transport mechanism, and shifting the burden of error recovery to the software of the attached processor greatly simplifies the hardware required.

The existence of two varieties of serial links is rather unsatisfactory, and arises because increased performance can only be achieved by the provision of buffering. Recent advances in semiconductor technology in the form of high performance cascadable FIFOs have greatly reduced the cost of providing a buffering scheme. This, together with the simplification of the hardware protocols, will allow the development of a new serial connection common to both local links and remote links.

Work is proceeding in a number of areas. An IEEE-488 interface is being built which will allow machines with this bus to be connected to Centrenet. Work has started into the transmission of voice and video traffic over Centrenet. As a result, one of the bits in the control field of the Centrenet packet now allows the receiving device to interpret the rest of the packet as it chooses. In this

manner datagram activity is supported by Centrenet. The transmission of voice over the network has been successfully demonstrated, and work has started into a new interface to a port-card which will allow the transmission of video data.

This paper has concentrated on the hardware aspects of designing a high-performance LAN. This does not mean that the software issues have been neglected. Studies (Bondi and Jackson (12)) have been initiated into the means by which Centrenet protocols may be matched to the layers of the OSI model (13). Work in this and a number of related areas, is still continuing. and will be published in the near future.

Acknowledgements.

The authors would like to thank Professor D. B. G. Edwards for provision of funding and support for the Centrenet pilot project. Some support has also been received from GEC and current work is being funded by the SERC as part of its Distributed Computing Systems programme.

References.

1. Morris, D., and Ibbett, R. N., 1975, 'The MU5 Computer System', Macmillan, London, England.
2. Lavington, S. H., 1975, 'A History of Manchester Computers', NCC Publications, Manchester, England.
3. Edwards, D. G. B., Knowles, A. E., Woods, J. V., May 1980, from a Mini-sized Computer', ACM Conference Proceedings, Seventh Symposium on Computer Architecture, 161-171.
4. Hopkins, T. P., 1980, 'An Investigation of Hardware Requirements for the Implementation of Communications within a Multi-Computer System', M.Sc. Thesis, Dept of Computer Science, University of Manchester, England.
5. Wilkes, M. V., and Wheeler, D. J., 1979, Symposium, Boston Mass., 47-60.
6. Metcalfe, R. M., and Boggs, D. M., 1976, CACM, 19 7 161-166.
7. Rawson, E. G, and Metcalfe, R. M., 1978, IEEE Trans. on Comms., 26 983-990.
8. Sikora, J. J, and Franke, D. C., 1983, 'A LAN Based on a Centralized-bus architecture', Proceedings of Localnet '83, New York, 1983, 147-157.
9. Lee, E. S, and Boulton, P. I. P., 1983 IEEE Journal on Selected Areas in Communications, SAC-1, 5 711-720.
10. Hopkins, T. P., 'The Design of a Local Area Computer Network'. Ph.D. Thesis, Dept. of Computer Science,

38 High performance LAN-Centrenet

University of Manchester, England - to be submitted.

11. Train D. A., 1982, 'An Optical Fibre Communications System for a Campus-Wide Local Area Network'. Ph.D. Thesis, Dept of Computer Science, University of Manchester, England.
12. Bondi, D, and Jackson A. R., 1984, 'Low Level Centrenet Protocols'. Internal Report, Department of Computer Science, University of Manchester, England.
13. ISO, 1981, ISO/TC97/SC16 Data processing - open systems interconnection - basic reference manual, Computer Networks, 5 81-118.

Chapter 4

Imperative languages in distributed computing

Richard Bornat

4.1. INTRODUCTION

There isn't very much distributed computing about - certainly I found less than I expected to while researching this review. To the programmer 'distributed computing' is still a problem, not yet a solution. We can't use 'distributed computing' as a tool or a description device in solving our programming problems in the way that we can use, say, recursion, repetition, conditional choice - or even concurrency. Instead it is a puzzle set for us by others, inspired mainly by advances in hardware design and production.

The problem is to control concurrency in a system which includes several processors. This is by no means a new problem, but solutions to it began to be practically important when it became cheap to print processors on silicon with very little labour input. It seemed immediately possible to use concurrent programming techniques to make a collection of cheap slow(ish) printed processors perform as fast as an expensively soldered and wired conventional machine. Networks of machines have existed for years, especially in finance and banking: cheap hardware made it seem reasonable to build smaller-scale networks which would behave as a single 'system'. Our experience so far seems to show that these things are certainly possible but are surprisingly difficult to bring into use.

Somehow or other 'imperative', when attached as an adjective to 'programming', seems to have acquired something of the meaning of 'pragmatic' or 'practical' - though adherents of other programming traditions might choose less flattering adjectives to describe our eager adoption of other people's objectives. We take it as the task of the language designer to help programmers solve problems which the 'real world' - of hardware designers, in this case - has thrown up.

Different uses for concurrent action could be expected to point different directions in language design. At the two ends of the hardware spectrum there are the multiprocessor and the geographically distributed network. The multiprocessor - in which processors share memory and

may share an input/output interface - is intended to run the parts of a single program concurrently; the problem is to make the processors behave together as if they formed a single 'machine'. The geographically-distributed network - many machines with no shared memory and non-reliable communication hardware - is intended to provide a reliable computing service in several places while sharing information and hardware resources; the problem is to permit each machine to use the rest of the network as if it is an extension to its own capacity.

In practice language development has been held up by difficulties of implementing mechanisms of communication. Few languages have contributed much in the way of program-structuring ideas and most designers seem to have concentrated on how to balance the advantages of implementing particular mechanisms against the costs of doing so. Much of my review, therefore, concentrates on those mechanisms and on the restrictions and caveats in language design, often put there for the sake of an efficient implementation.

4.1.1. Program structuring

Most of the languages reviewed here (occam (3) is the most important exception) are developments of Pascal (27). In Pascal, as in any 'procedural' language, the important units of execution are the program, the procedure and the instruction. At a particular level of description a program execution is a collection of procedure executions. If an instruction calls for a new procedure execution then one is created and starts operating; meanwhile the calling execution pauses and waits for the new one to terminate. Thus only one procedure execution can ever be operating at one time. Even coroutine languages, like Modula-2 (15) or BCPL under TRIPOS (14), are sequential according to this model (although the procedure executions don't wait in line in quite the same way there can only be one of them operating at any time).

In concurrent languages more than one thing can happen at one time. Every language uses a different terminology to describe those executions which can be operating concurrently: I have chosen to use the word process. A concurrent program execution may contain more than one operating process execution.

One particularly simple extension of the procedural model identifies a process as a collection of procedure executions. This is the approach of Martlet (9), of Pascal-m (4) and of Conic (7): processes in such languages are each the same as an ordinary sequential program except for the way in which they communicate with their environment. There is a simple hierarchy of executions from program through process and procedure to individual instruction executions.

An alternative approach allows an operating execution to 'fork' into several concurrent executions which later 'join' back into one. This is essentially the approach of PascalPlus (26) and Ada (10). There is no longer a simple

hierarchy of executions: instead procedure and process definitions can be seen as alternative structuring mechanisms.

Occam (3) takes this second approach farther than any other of the languages reviewed here. The instruction execution is the basic unit, just as in the more conventional languages, but instructions can be composed as easily in parallel as in sequence. Iteration and guarded commands are alternative ways of composing executions.

In scoping and visibility rules most of the languages are rather old-fashioned. Martlet, Path Pascal and Pascal Plus use Pascal scoping; occam uses straightforward hierarchical scoping. Ada is of course more adventurous but its 'package' and 'generic' mechanisms are too well known to require discussion here. Conic and Pascal-m have a 'module' mechanism which constrains the interconnection of processes. Modula-2 has modules, BCPL, Coral and C are just ordinary procedural languages.

4.1.2. Communication mechanisms

It is normal to make a distinction between language mechanisms and implementation devices. In principle any of the mechanisms used in any of the languages could be implemented in terms of any of the others. But designing the implementation of particular mechanisms - i.e. finding a device which fits a chosen mechanism - has often affected the choice of mechanism and has been an important influence on the design of most of the languages reviewed. To understand the designer's choice of mechanism it is usually necessary to understand what devices have been invented.

The presently popular language mechanisms are shared memory (including monitors in PascalPlus and objects in Path Pascal), Remote Procedure Call, buffered message-passing and synchronised message-passing. Each of these mechanisms involves the transmission of information, though some procedure calls in Path Pascal can be used as pure synchronisation operations.

Shared memory, or more strictly shared information, is the oldest of the mechanisms. If the 'naming space' of several processes simply overlaps so that the effect of an assignment by one is visible to the others then communication between them is obviously possible. It is always essential to ensure that assignment and access actions never overlap because in general they can't be made atomic operations of the underlying hardware. Both PascalPlus monitors and Path Pascal objects wrap shared memory object declarations with definitions of procedures which must be used to update or access the value of those objects; then they control the use of procedure executions so as to constrain or eliminate overlap. The monitor mechanism makes critical procedure executions effectively atomic, while Path Pascal allows the description of a degree of concurrency between executions. Shared memory communication isn't synchronised, though processes must often wait to be permitted to create or to complete a procedure execution. Communicating processes aren't paired

42 Imperative languages

- information is genuinely shared between whatever process puts it there and whatever processes inspect it.

In a Remote Procedure Call one 'sender' process provides the parameters for a procedure execution created by another 'receiver' process. The sender communicates with the receiver by providing parameter information and by receiving results. The receiver communicates with the sender by sharing memory with the procedure execution. The two process executions each wait for the completion of the procedure execution. Processes are obviously paired and communication is synchronised in that a sender is delayed until the receiver offers to create the execution; conversely a receiver is delayed until there is an appropriate sender.

In buffered message-passing a sending process fires off a message and doesn't wait for a receiver to be ready to accept it. Conceptually the receiver holds a queue of messages sent to it but not yet inspected. Sometimes there may be a priority scheme to order messages in the queue and often a receiver may be able to detect whether its queue is empty and hence may be able to choose whether or not to suspend execution so as to wait for a message. Processes need not necessarily be paired: there can be more than one receiver process for a single send, as in Conic (7,29), which will produce a limited kind of 'broadcast' effect.

In synchronised or unbuffered message-passing a sender must wait until a receiver is ready to accept the message; conversely a receiver must wait for a sender. This kind of message-passing is synchronised and paired though typically a receiver may choose between many potential senders before communication occurs.

Buffered message-passing can cause a problem of memory starvation because each message sent requires buffer space. It is easy to see that over a period every process in a system might send more messages than it receives and the system might then deadlock because buffer space is exhausted. Synchronised communication - either message-passing or Remote Procedure Call - has the advantage that sending a message usually needs no storage apart from that allocated in the participating processes: the compensating disadvantage is that it can be difficult to synchronise processes on different machines, though language restrictions can help to reduce this difficulty by restricting the ways in which synchronisation can come about.

4.1.3. Making connections

Languages which don't exclusively use shared information must allow senders to name receivers and/or vice-versa. The issue is how to name a process execution and whether to do so directly or indirectly.

In all the languages reviewed which use Remote Procedure Call a sender names the receiver process directly, names the procedure to be executed within the receiver and gives argument values; a receiver offers to create an execution for any sender naming the correct procedure. It is unnecessary for the receiver to name the sender and the

connection pattern in every language is therefore any-to-one. The receiver's offer may be continuous, as in Ada 'task procedures', or intermittent, as in Ada 'task entries'. In principle a sender might name a receiver indirectly, giving only the procedure name and argument values so as to be paired with any receiver offering to create an execution of that procedure - i.e. any-to-any connection. This isn't a facility of any of the languages reviewed but something like it has been provided in the MINAS operating system (17).

All the message-passing languages reviewed here use some form of indirect channel addressing (called 'ports' in Conic, 'mailboxes' in Pascal-m, 'channels' in occam). Communication is sent to or received from a channel and the partner is whatever process makes the opposite communication on the same channel. Channels can be typed to constrain the messages which may be sent or received through them. A channel mechanism makes it possible for a sender or a receiver to select between different sets of potential partner processes. In occam the decision to use channels may have been made because the design of the language makes it impossible to name a partner execution. In Conic and in Pascal-m process executions can be named directly but in each case the designers adopted a channel mechanism to facilitate dynamic reshaping of the process-connection pattern.

Indirect naming, whether by channels or by procedure names, has an analogy with information-hiding. If sender and receiver have a client-server relationship, as they often do, then directing communication to a process means that the programmer who designs the client process must have more information about the server than is strictly necessary. Directing communication through channels gives far more potential for modular programming.

Both Conic and Pascal-m use a form of module construct to control the connection of processes to channels. Conic uses a configuration description to link ports on different modules and, potentially, to unlink them again. Pascal-m modules use an import and export mechanism to control the visibility of mailboxes across module boundaries.

4.1.4. Non-determinism

Non-determinism is inherent in a concurrent program with several operating executions because the program text specifies only a partial order of program events. Non-determinism arises when processes are connected any-to-one because a receiver cannot control which of all the potential senders will succeed. Similarly in one-to-any connection a sender cannot control which receiver will accept and in any-to-any connection neither partner has control.

Non-determinism can also arise in languages using synchronised communication from the program construct called a guarded command, first introduced as a communication construct in CSP (26). A guarded command allows a process to make several simultaneous offers to communicate. Each

44 Imperative languages

offer is written so as to label, or guard, an instruction; the first offer to succeed triggers execution of the instruction it guards and all other offers are cancelled. The guards themselves can be protected by Boolean formulas to control which of the offers are actually made in a particular execution of the guarded command. So, for example in occam (3) a guarded command with three alternatives could be written as follows:

```
ALT
  x /\ y<0 & c1?f
    I1
  c2?g
    I2
  z=3 & c3?h
    I3
```

Communication is offered on channel c2 by every execution of this command, on channel c3 only if z=3 and on channel c1 if x is true and y is negative. Whichever communication happens first causes the instruction following it to be executed and then the guarded command completes.

4.2. IMPLEMENTATION INFLUENCES

Pseudo-concurrency, in which a concurrent program is executed on a single processor, is rather easy to implement. It's easy because on a single processor only one thing happens at a time, so that implementing a virtual machine which provides concurrent communication primitives becomes rather like implementing a coroutine machine.

On concurrent hardware the trick isn't so easy, and here we can distinguish two distinct difficulties. The first is that there is no global clock. Since information travels between machines at a finite speed it is impossible to decide in general whether an event in one machine occurs before or after a different event in another. The problem arises naturally when deciding whether an offer of communication made from one machine has been accepted in another and affects the design of languages using guarded commands and also those which use so-called 'time-out' limits on the duration of a communication attempt. A second difficulty arises because some distributed systems are organised so that part of the system may break down while the rest is capable of continuing without it.

4.2.1. Timing problems

If several processes make offers of communication to each other at about the same time there is a problem of arbitrating whatever conflicts arise. If process A offers to communicate with B and C, B with A and C, C with A and B then at most one of the three possible pairings can take place. The problem isn't too hard to resolve on a uniprocessor because there can be a single arbiter and while it executes the processes don't. With several processors -

one each for A, B and C, say - the difficulty is that none of them knows the current state of any of the others, but only their states in the recent past.

It is possible to avoid the timing problem and recreate the communication characteristics of a uniprocessor implementation by having some controlling hardware device such as a section of shared memory or an arbitrating processor which acts as a global synchroniser. So, for example in the implementation of Martlet (9) each multiprocessor station contains a shared memory, a control processor and a number of application (task) processors. When a process, running in a task processor, offers a communication a marker is deposited in the shared memory and the task processor waits. The control processor arbitrates its next action - typically, it schedules some other process for execution.

Synchronising hardware can obviously be a bottleneck in execution if too many processes consult it at once or if it takes too long to consult. The hardware is also uniquely important to the system as a whole: if it breaks the whole system stops.

An alternative is to establish some protocol conventions by which autonomous processors negotiate to decide which communication to perform next. This becomes difficult if an offer once made can easily be withdrawn. If an offer from process A, running on one machine, is sent to process B on another machine that offer will take some time to reach B. B's acceptance, if the offer is eventually accepted, will take some further time to return to A. During that time process A may have received alternative, perhaps more attractive, offers to communicate or the interval may be so long that it simply decides to 'time out' (i.e. to give up the attempt). Then B's acceptance may have to be rejected.

Without care in the design of a protocol there is the possibility of 'livelock' in which processes never agree. Taking the example of processes A, B and C once more: suppose that A decides to accept B's offer (in effect withdrawing its offer to C), B decides to accept C and C decides to accept A. Each process will send an acceptance to one which has already decided to reject its offer and there will be no pairing and no communication. Even if they start all over again from the beginning the situation can be repeated indefinitely.

Protocols have been published which use process-numbering to avoid livelocked cycles of processes - for example recently (30) - but they are relatively complicated to implement and expensive to operate. Most language designers have restricted their languages to avoid the problem. If only one sort of communication - typically receive - can be offered in a guarded command then only that sort of communication can be withdrawn. The other sort of communication - i.e. the send - must be offered unconditionally, without even a time-out alternative. Then a process executing a guarded command can examine the send offers made to it without any need to negotiate: whichever it decides to accept will still be in force when it accepts.

Yet another alternative solution is to ignore the problem

46 Imperative languages

and to push it back onto higher-level software - for example in (7,29) and (19) an offer to send which is withdrawn by time-out may already have succeeded, undetected by the sender.

4.2.2. Hardware failure

The difficulty of programming a system which may partially break down concerns some language designers. In a uniprocessor system if one component stops the whole machine stops. Then programmers don't have to consider what to do if part of the machine, carrying part of the program, should break down or be isolated from the rest by communication hardware failure. If any computing system, distributed or otherwise, is seen as running a single concurrent program and doing it concurrently because that is just a nice way to program a solution, then component failure can be treated just as on a uniprocessor. But if a system is made up of potentially autonomous machines, or if concurrency is being used to provide a reliable computing service, then component failure must not stop the system. (7,29), (4), (17) and (19) address this issue in different ways.

4.2.3. Shared memory

Shared memory seems a rather unlikely implementation mechanism for a geographically distributed system. The monitor, the most commonly-used mechanism for interleaving access and assignment in a pseudo-concurrent implementation, seems to have no intrinsic advantages over the Remote Procedure Call. Path Pascal provides an alternative interleaving and synchronisation mechanism which does claim some advantages over the RPC so perhaps shared memory may have a future after all. The UMIST experience of implementing PascalPlus (23) leads that group to conclude that shared information rather than shared memory is the useful linguistic notion.

4.3. THE LANGUAGES

In this section I consider how some of the languages which have received a significant implementation effort, or which are used to a considerable extent, match up to my Procrustean criteria. Non-mention of a language here shouldn't be taken as a mark of criticism in itself, merely as evidence for the damage caused by my own myopia.

I concentrate mainly on the communication mechanisms which the languages use. Most of the languages are developments of Pascal in any case, so that their program-structuring facilities are broadly similar. Naturally I try to point out important points of difference.

4.3.1. Ada and Martlet

A group at the University of York (11,12) have used Ada (10) as the basis of the PULSE distributed computing project; a group at the University of Sussex (9) has incorporated some of the Ada language into Pascal, producing Martlet. Because the languages are so similar I discuss them together, giving most attention to Ada. Martlet is Pascal without i/o and file types, plus some of the Ada communication mechanisms and some of the its exception mechanism. Crucially Martlet does not include Ada conditional and timed entry calls and it prohibits pointers in arguments to entry calls - that is, it leaves out just those features which may make Ada unsuitable for use as a distributed computing language.

As to Ada then: it is difficult to give a fair review to a language whose main claim to acceptance must be the active support of one powerful computer purchasing organisation, but I shall try. The size of the language means that I can review only a tiny part of it.

The process-structuring construct is the task. In Martlet a program is a set of tasks, so that in effect the language has the program, process, procedure instruction hierarchy described in section 4.1. Ada tasks can be created more freely, in procedures, packages, blocks and so on. The creating execution can't terminate before the task it creates have also terminated, so the mechanism is a sort of fork-join.

The communication construct is remote procedure call. Task definitions define 'procedures' and 'entries': procedures may always be called, entries are in effect procedures which can be called only when the task is executing an appropriate accept instruction. Task type definitions have two parts: the header declares the procedures and entries which the task type provides, the body gives definitions of the actions of those procedures and entries. So, for example

```
task A is
  procedure P(a: out integer);
  entry U(b:in y);
  entry V(c:out z);
end;
```

defines the interface of a task-type A. The body of A shows how it permits sequences of UV and VU pairs; procedure P returns the number of pairs that have completed. The first of the accept entries for V is inside the accept for U; likewise the second U is inside the second V. The accept entries act like procedure declarations and the scopes nest in the normal way.

48 Imperative languages

```
task body A is
  count: integer:=0;

  procedure P(i:out integer) is begin i:=count end;

begin
  loop
    select accept U(q: in y) do .. q is in scope ..
      accept V(r: out z) do
        .. q and r are in scope ..
      end V;
    end U;
  or   accept V(s: out z) do .. s in scope ..
    accept U(t: in y) do
      .. s and t in scope
    end U;
    end V;
    count:=count+1;
  end loop;
end A;
```

Other tasks can now communicate with A using procedure calls such as A.P(j) - which always succeed - or entry calls such as A.U(f) and A.V(g) - which will only succeed in this example if used at the right time. Because the accepts are nested the process which succeeds with the first U entry or the second V entry is held in synchronisation until the matching entry in the UV or VU pair is made by another process. This extended synchronisation is a peculiar feature of the remote procedure call mechanism: the select statement could be rewritten to use procedure calls more like message operations, though with different effect:

```
uc:=0; vc:=0;
loop
  select when uc<=vc =>
    accept U(q: in y) do .. q is in scope ..
      uc:=uc+1;
    end U;
  or when vc<=uc ->
    accept V(r: out z) do .. r is in scope ..
      vc:=vc+1;
    end V;
    count:=count+1;
  end loop;
```

As defined in Ada the communication mechanism is adapted to a shared memory implementation: pointer ('access') values can be passed as arguments in remote procedure calls and there are conditional and timed versions of the entry call

```
select <entry call> else <instruction sequence> end;
```

- the <instruction sequence> is executed if the entry call can't be immediately accepted

```
select <entry call> or delay <interval> end;
```

- the select terminates if the entry isn't accepted before the interval expires

Use of pointer values requires some other form of communication underneath the rendezvous mechanism - shared memory seems the most likely candidate. The language provides no mechanism to restrict or control simultaneous access or assignment to memory shared in this way though the language definition, under the heading of 'aliasing', warns the programmer to take care. The conditional entry call would seem to be of doubtful utility inappropriate in a geographically distributed implementation because the time it takes to find out whether a receiver is ready to accept an entry call will often be comparable with the time it takes the receiver to change state from unready to ready. The timed version of the entry call is difficult to implement for the reasons given in section 4.2.1 above.

The naming mechanism of Ada forces sender to name receiver task in an entry call. An important criticism of both languages must be that this makes the process connection pattern much too fixed. A more abstract interface, some kind of port mechanism perhaps, would make it possible to decouple the request for service initiated by a sender from any particular receiver task.

The use of procedure call as a communication mechanism ties sender and receiver together for the duration of the call. It has the advantage that the receiver can give replies to the sender via **out** parameters in the entry call without knowing the sending task identity. It has the further advantage that no process-switching is required to transmit a reply from original receiver to original sender. It makes it impossible, though, to receive a reply from any but the process to which the entry was made. And in (12) the York group show how difficult it is to program a receiver which may not want to make an immediate reply to every send.

Using procedure calls in the style of messages might overcome some of these problems but would require a receiver to know the sender's identity, which can be tricky to arrange even with shared memory and pointers to tasks.

Ada as it stands seems to be a pseudo-concurrent language or at best a language for multiprocessors with shared memory. I'm sure it would be possible to implement the full language for a more loosely coupled system but I'm equally sure that it wouldn't be easy and that the implementation wouldn't be particularly efficient. Without the nasty bits an implementation is much more feasible - as shown by the Martlet implementation, which has been running for some years.

Either language makes it possible to write fully type-checked programs for a multiprocessor machine, whether network connected or shared-memory. They don't aim at the construction of multi-program systems for a multi-computer network (a fact, not a criticism: those are distinct aims and certainly no language reviewed in this paper solves both

problems). Fixed process connection patterns make it difficult to see how to cope with any sort of partial hardware failure or dynamic network configuration.

The PULSE project aims at the construction of an operating system for a multi-computer network, thus using Ada for a purpose to which it is manifestly unfitted. As a result the York group have had to resort to the standard subterfuge: they use Ada as a pseudo-concurrent language within each station of the network and use buffered message-passing between stations. To communicate with a distant station a process makes an entry call on one of several special tasks, called 'Mediums', which are in effect network handlers. The types of message which can be passed between stations are necessarily fixed - one for each kind of special task - and can't be varied at the whim of the programmer.

4.3.2. Conic

Conic (7,29) is a language designed for a particular problem area, a certain kind of physical-process control. It is a development of Pascal. It is in use at Imperial College and being evaluated by more than one industrial backer for commercial use. It was partly funded by the National Coal Board for use in mines where horrible things happen to electrical machinery - hence perhaps the designers' attention to network reconfiguration and communications hardware failure.

In a Conic network each machine is reasonably reliable but the system as a whole is less reliable: it may be reconfigured at the whim of its human controllers or it may partially break down. Some machines observe measuring devices and report their value at intervals to other machines. Some machines control physical devices according to a defined tactical program. Some machines have strategic responsibility, altering the tactics of others according to information received from devices or from other machines. Some machines have all three functions.

A crucial point is that the machines in a Conic network should be viewed as potentially autonomous and should be programmed to continue alone if they become isolated, either by hardware failure or by reconfiguration of the network. An isolated observing machine should then continue to observe and continue to try to send its reports until it is re-integrated into the network. An isolated controlling machine can do something useful if it obeys the last tactic it was ordered to follow. An isolated strategy machine must sit on its computational thumbs until the fault is repaired. The kind of messages exchanged - status reports and change-tactics commands - are by their nature idempotent, so repeated transmission is a reasonable behaviour. Unreliability of hardware and the impossibility of completely ordering events are burdens which seem to be bearable within this problem domain.

The process-structuring constructs are the module and the task. A Conic program is a collection of modules, each of which is a collection of tasks. A task is like a

Pascal program in that it consists of a collection of procedure executions, only one operating. Tasks within a module share an address space but can only share memory if they pass messages to each other which contain pointers. Pointers are prohibited in messages between modules.

Tasks communicate through ports: a task definition declares certain ports and all its communication must be directed to its own exitports or from its own entryports. A separate configuration specification describes how the ports of one task are linked to those of another. So, for example, a simplified Conic module which reports a temperature setting at defined intervals and which can be ordered to change the interval:

```
task thermometer(i:interval);
  entryport interval: integer;
  exitport report: temperature;

begin
  repeat
    select receive i from interval => skip
    or timeout i => send temp to report
  end
  until false
end;
```

The select instruction is a guarded command which allows several receive alternatives: here it is used trivially with a single timeout alternative.

The configuration specification can now connect the thermometer task to another task, or even to two different tasks, provided that the port types match. Tasks communicate if they address matching communications to ports joined by the configuration description. Potentially, but not yet in practice, the port connection pattern can be altered while the system is running. Separation of configuration description from operational description is claimed to make for modular programming and certainly seems to do so.

There are two kinds of communication ports. One is a notify port, clearly designed for the reporting and broadcasting of status information. Both ports of the thermometer example above are notify ports. The connection pattern is one-to-many: a single notify exitport can be connected to several notify entryports and all those entryports will receive any message sent. The messages are buffered, but the splendidly eclectic design of the language avoids the problems of buffer exhaustion and the overheads of buffer allocation by making each receiver allocate space for a fixed number of messages. So, for example, a module which was connected to the thermometer module might have declared an entryport

```
entryport heat: temperature queue 5
```

and be able to buffer five incoming messages. New messages just overwrite the oldest ones if necessary, so that a

52 Imperative languages

receiver can see only the newest messages in a sequence sent to it, and only the latest message if it allocates a single buffer, which is the default. It seems that accurate up-to-date information is more important than a complete history in the world of process-control. Senders are never suspended on a notify port: receivers may be if the queue is empty but a process can discover whether the queue is empty or not by using a special form of select instruction.

The second kind of Conic port is the request-reply which has two types, one for the request and one for the reply. So, for example

```
exitport faster: speed reply boolean
```

allows a process to send a message of type 'speed' and receive a boolean reply, indicating perhaps whether a distant process was able to alter some device to work at that speed. In operation a sender sends a message to an exitport and waits for a reply or a failure through the same port:

```
send <message value> to <exitport>
  wait <reply variable> => <instruction sequence>
  fail <timeout interval> => <instruction sequence>
```

If it gets a reply it executes the instructions following wait; if the timeout interval expires, or the communication hardware breaks, or the exitport isn't linked to any entryport the second instruction sequence is executed instead. The program can discover the reason for failure if necessary.

A receiver accepts messages, constructs a response and sends it back through the same port:

```
receive <message variable> from <entryport>
  ... <instruction sequence> ...
reply <reply value> to <same entryport>
```

Communication is synchronised and unbuffered: a sender can't proceed until its offer is accepted or until its delay expires or the connection seems to fail. Request-reply receive instructions can be used in a guarded command.

Timeout occurs in a sender if the reply doesn't arrive in time (cf. Ada in which the timeout occurs if the entry isn't accepted in time), which by no means implies that the message wasn't sent or wasn't accepted. Evidently a message might be sent, actually get through, be processed and a reply be transmitted but the sender timeout before the reply arrives. Such delayed replies are by definition discarded, and programmers are warned that a timeout doesn't mean a message wasn't sent. Likewise, communication failure may have occurred before, after or during message transmission. In the world of process control this sort of uncertainty is apparently acceptable.

Conic is too ad hoc for my taste, but I am impressed with the way that it tackles the practical problem of how a language should treat reconfiguration of a network, failure

of the network hardware, failure of a receiver to keep up with its senders and the impossibility of synchronisation of processes on different machines within a particular universe of application. Conic is fine if you want to program the sort of system its designers envisage and it certainly allows you write type-secure programs for process control networks. If you are running on reliable hardware then obviously some features of the language become less useful.

Because communication is indirect through ports then, if modules can be written to operate autonomously when they become isolated, it is potentially possible to reconfigure a running system to cope with partial hardware breakdown. I understand that Conic will continue to be developed in that direction.

4.3.3. occam

I feel impelled to say that occam (3) is a beautiful language. Indeed nothing so lovely has happened in language design for ten years. It is so nice that I feel irresponsibly light-hearted about it. So, some irresponsible jokes: why do I have to write occam with a small o? - because it's a natural naming-progression from FORTRAN through Pascal. And why must I write each instruction on a fresh line? - so that the compiler-writers can measure the compilation speed in lines per minute and at last compete with IBM FORTRAN on level terms (as an ex-compiler-writer myself I half believe that one).

Seriously, though: occam isn't based on Pascal, which is a fine relief for this reviewer. The language is amazingly simple: the official definition is about fifty pages, most of which is taken up by blank space and example programs. Most of all it resembles a productive cross between BCPL (24) and CSP (26). It inherits from both its parents elegance and simplicity of design and an intense concentration on ease of implementation within its chosen area. Everything is pared down to ensure that occam programs use a fixed amount of space: this is a language targeted at arrays of microprocessors running a concurrent program with the minimum of 'underlying mechanism'.

So occam prohibits recursion because, when there are several programs running in a machine, recursion needs an underlying mechanism which can rearrange the allocation of memory whenever execution stacks collide. It uses synchronised communication because that mechanism requires no buffering, and hence no global buffer-allocation mechanism. Its channels are one-to-one connections because that means no queuing - again, no underlying mechanism required. Its guarded commands have receive guards only because that simplifies implementation of synchronisation. The number of processes in every parallel composition, and the number of guards in every guarded command, is fixed and can be counted by the compiler and therefore it can allocate fixed space to every process no matter how complex its execution structure.

You might expect that such concentration on efficiency would produce a linguistic straitjacket. Not so: what you

54 Imperative languages

get instead is a language in which parallel composition, because it is so efficient, can be used as freely as sequential composition. So every instruction-execution is an occam 'process' and you write each basic instruction on a separate line to emphasise the point. There is no bracketing: instead you write a composition-phrase like SEQ, PAR, ALT or WHILE on one line and the instructions it composes on following lines, indented to show their dependence. Lifting an example from the occam manual:

```
CHAN c[n+1]:
PAR i = [0 FOR n]
  WHILE TRUE
  VAR x:
  SEQ
    c[i]?x
    c[i+1]!x
```

- which describes the parallel composition of $n+1$ processes, the i th of which takes input from channel c_i and passes it to c_{i+1} . Together they form a sort of n -stage shift register. Longer examples are equally easy to parse: here is one of my own, which is part of a protocol description

```
CHAN alldone:
PAR
  SEQ
    SEQ i = [1 FOR n]
      ms[i]!CNCL
    alldone!ANY

  VAR finished, x:
  SEQ
    finished:=false
    WHILE NOT finished
      ALT
        ALT i = [1 FOR n]
          g[i] & mr[i]?x
          g[i]:=false
        alldone?ANY
        finished:=true
```

- a composition of two parallel processes, one of which sends n messages on channels ms_i and then signals to the other that it has finished; its partner waits in a guarded command listening to some of the n channels mr_i and simultaneously awaiting the termination signal.

The language has straightforward hierarchical scope - no fancy module syntax here. There is a top-level configuration language, which allows processes to be allocated to individual hardware processors within the system and the inter-processor channels to be declared. Processes are then hierarchically decomposed: if a process forks into several parts then they all run on the same machine, as you would expect.

Because the language has hierarchical scope the

name-spaces of two processes composed in parallel will overlap. Indeed they must overlap if they are to communicate over a shared channel, as in both the examples above. But sharing memory objects (variables) is trickier. Processes may freely access a shared memory object provided that none of them assigns a value to that object, says the language definition. No compiler can enforce such a restriction completely, because of the tricks a programmer can play with array indices, so it is just an exhortation requesting the programmer to behave properly, rather like the Ada exhortation to avoid the bad effects of aliasing. Equally the restriction to one-to-one channel connections is an unenforceable exhortation. Thus occam inherits from BCPL not only simplicity but also something of its character as a 'high-level assembly code', a sharp tool with which you can easily cut yourself. This is perhaps an inevitable characteristic of any systems implementation language.

With or without Justice in This World, I would vote occam the Language Most Likely to Succeed. I fear only that it might be overtaken, as its ancestor BCPL was before it, because it is word-based although so much computing practice has to do with strings of bytes and byte-structures of various sizes. Occam allows vectors of words or of bytes, but makes no other concessions to data structuring: like BCPL it recognises only the vector. C (25) superseded BCPL for many reasons, one of which was certainly that C accomodates the byte-addressing structure of modern machines in a way that BCPL doesn't and C gives some superficial recognition to data structuring. It would be a shame if occam, like BCPL, is superseded by something as nasty as C and for the same sort of reason.

4.3.4. Pascal-m

I shouldn't say too much about this language (4) because it is partly my own invention. Another development of Pascal, like Conic it has processes which are collections of procedures. It has a module syntax which groups process and channel declarations and which restricts the visibility of channels and thus the initial interconnection pattern.

The language's main innovation is the mailbox: a form of any-to-any channel connection. Mailboxes have identifications (effectively addresses) which can be sent in messages so that the initial interconnection pattern, set by the way in which modules import and export mailboxes, can be dynamically reconfigured. We succumbed to the temptation to make processes dynamically invocable - a process can create another almost as easily as it can call a procedure within itself. Examples of programs in the language are given in (31).

Pascal-m was designed to make a certain kind of programming easy, with rather less than half an eye kept on the difficulties of implementation. So mailboxes are any-to-any channels, guarded commands allow both receive and send guards and mailbox identifications can be sent in messages. These features together make it easy to describe solutions to some problems, but impossible to give Pascal-m

message-passing a simple implementation. Because mailboxes are any-to-any channels, there must be some queueing of offers to communicate. Because guarded commands are bidirectional offers once made can at any time be withdrawn and an expensive protocol is required. Because mailbox identifications can be sent in messages it is impossible to determine the set of processes which can communicate over a mailbox.

A pseudo-concurrent implementation has existed for several years, but true concurrency on distributed hardware has so far eluded us - in part because we wish to produce an implementation which is both distributed and fair. We have for some time known of one protocol which would permit a distributed implementation and recently we have developed a second; we are still determined to 'distribute the language'.

Our intention was to provide a language for networks which require an expanding population of processes and even of process types. So in a network of office workstations we envisaged that a novel application program could be compiled and incorporated into a running system, as securely type-checked as if it had been compiled at the same time as everything else. Mailbox identifications which can be passed in messages make this kind of extensibility potentially possible. We didn't take enough account, perhaps, of the problems of machine failure and network failure.

I think, though, that I am falling over backwards not to be seen to be favouring my own work. Together the group at QMC has written a filing system in Pascal-m, a UNIX-like operating system and several sample user-interface systems providing a multi-window screen interface. The project continues with industrial support from Texas Instruments. The problem of programming Flexible Manufacturing Systems seems to be an application area where problem descriptions are complicated, message rates are fairly low and flexible extension of running systems would be very useful. We think the language has a useful future.

4.3.5. Path Pascal and PascalPlus

PascalPlus (23) was a very early concurrent language. Its communication mechanism is the monitor and its obvious implementation technique is shared-memory. Its current guardian tells me it deserves little more than a footnote in this review. Nevertheless it has had a true distributed implementation at UMIST (32) and another is underway at Sheffield (33). The UMIST implementation used a shared memory and control processor on a Cambridge Ring - i.e. a sort of geographically distributed version of the Martlet multiprocessor (9).

Monitors are a means of interleaving assignment and access to shared memory objects: control is by making procedure executions mutually exclusive. Path Pascal (5) uses a similar notion. Executions need not always be exclusive but the amount of concurrency can be controlled. So, for example, given definitions of procedures put and get

```

entry procedure put(i:t);
begin b[in]:=i; in:=(in mod n)+1 end;

entry procedure get(var j:t);
begin j:=b[out]; out:=(out mod n)+1 end;

```

the path expression

```
n:(1:(put); 1:(get))
```

expresses the restriction that there can be up to n concurrent executions of (put;get) but that within that concurrency, executions of put are mutually exclusive as are executions of get. This gives all the information required to control insertions and deletions of an n -place buffer. The procedures can be defined more simply than usual, because the underlying mechanism counts to make sure the buffer is never overfull or less than empty.

So far as I know the language doesn't yet have a distributed implementation, though one is envisaged (6). Clearly it would be possible to implement a Path Pascal program so that each object was on a separate processor of a network, or to implement server processors written in Path Pascal within a heterogeneous network. The path expressions give some of the implementation advantages of occam, in that the space requirements of an object can be worked out in advance in many cases.

4.3.6. Programming in unhelpful languages

All of the languages dealt with so far have provided some facilities to help with concurrent programming - guarded commands and type-checking of messages, for example. A good deal of distributed computing - in fact practically all the non-experimental distributed computing in this country is done in languages which give little or no help to the programmer. For example: telephone exchanges, which are typically collections of machines, each a multiprocessor, are programmed in CORAL (1,2). The Cambridge Ring installation at the University of Cambridge is programmed in BCPL (14). The Newcastle Computing Laboratory network is programmed in C (20) as is the network at Strathclyde (17). Kent use both BCPL and C (33). Oxford program in Modula-2 (15).

Every one of these installations in effect uses the same solution. A coroutine language provides a kind of pseudo-concurrency on a single processor - Modula-2, BCPL under TRIPOS, Post Office CORAL, C under UNIX (C gives no help at all but UNIX(35) gives pseudo-concurrency). Messages which are all of a single type - byte-sequences in every example except in the telephone exchanges, which use a fixed record type - are transmitted between machines. Typically on each machine designated processes handle outgoing and incoming messages: the use of pseudo-concurrency means that the incoming-message process can be scheduled to operate

whenever one arrives.

The York PULSE project (11), although I treated Ada as a genuine concurrent language above, is really another example of a programming system where a concurrent language is used to give pseudo-concurrency on a single processor and messages of fixed type are transmitted between Medium processes (effectively message buffers) on different machines.

UNIX doesn't lend itself well to message-passing, but procedure call is the fundamental means of communication with the operating system and hence with other processes in a conventional single-processor implementation. Both at Newcastle (20) and Strathclyde (17) there have been implementations of Remote Procedure Call which allow UNIX systems to be linked by a network - initially a Cambridge Ring in each case. The Strathclyde implementation provides interprocess communication across machine boundaries, with port descriptors as an indirect process-addressing mechanism.

It would be wrong to criticise these implementations because they didn't use a language capable of describing the 'program' which is running on their entire system, or to criticise the facilities they provide in the same way as those that are provided in concurrent languages. The problem they address is that of linking computers which are themselves self-contained systems, which must carry on running even though other parts of the system break and which must be removable from the system for hardware maintenance or software alteration. In effect the 'network' is a rather flimsy alteration to the environment. So far no language has gone very far towards a solution of this very real problem of loose, intermittent coupling.

4.3.7. Other languages

There are a number of languages which are being developed or are in use and which I can do little more than name. In some cases I haven't reported because they aren't used much; in others because they won't fit into my classification. DTL (22,28) is reported on elsewhere in this conference. Edison is under investigation at Sheffield Polytechnic (16,33) in a sort of competition with PascalPlus. Basix (18) was developed at the University of Newcastle in an attempt to extend the way in which the UNIX 'shell' handles file-hierarchies to other hierarchies, in particular hierarchically organised hardware and hierarchical scopes in programs. Lisp (8) was experimented with at the University of Bath on a multiprocessor. There is a development of Basic (21) which aims to provide 'real time' programming and could, say the authors, be extended to distributed concurrent programming: the reference gives a Basic version of the Conic pump-control program.

4.4. CONCLUSION

This review is not intended to be depressing or to condemn the current state of imperative language design in distributed computing. Superficially there don't seem to be many distributed programming languages in use and most of the distributed computing which is done seems to be either experimental or else done under UNIX. But in fact there are grounds for optimism. We have at least one excellent language design in occam, and a good deal of experience of implementing and writing programs in several more. We haven't reached our destination yet but the wagon is rolling and all its wheels seem to be going round (our mule was the DCS programme and since this is the final conference it can't pull the analogy any farther). The next five years won't be all downhill but I am sure that much of the hard work of the last five will begin to pay off.

REFERENCES

1. Park I.D.C., 1981, Post Office Electrical Engineers Journal, 74, 81-86
2. Loomes S.R., 1980, Post Office Electrical Engineers Journal, 73, 47-54
3. INMOS Ltd, 1983, occam programming manual, Bristol, UK
4. Abramsky, S. and Bornat, R., 1983, "Pascal-m, a language for distributed computing", in Distributed Computer Systems, ed. Y Paker, Academic Press, London, UK
5. Campbell, R.H. and Kolstad, R.B., 1980, "Path Pascal Users Manual", ACM SIGPLAN Notices Sept 1980, pp 15-25
6. Dowsing R.D. and Elliott R., 1984, "Implementation of Object Oriented Languages on Distributed Computing Systems", Proc of Conf. on Hardware to support Distributed Systems, Bristol, 1984
7. Kramer J., Magee J., Sloman M. and Lister A., 1983, IEE Proceedings part E, 130, 1-10
8. Marti, J. and Fitch, F., 1983, The Bath Concurrent Lisp Machine, in Lecture Notes in Computer Science 162, 78-90, Springer-Verlag
9. Grimsdale R.L., Halsall F., Martin-Polo F. and Wong S., 1982, IEE Proceedings Part E, 129, 63-69
10. "Preliminary Ada Reference Manual", ACM SIGPLAN Notices June 1979
11. Wellings A.J., Keefe D., Tomlinson G.M. and Wand I.C., 1983, "Programming Distributed Systems in Ada", Department of Computer Science report, University of York

60 Imperative languages

12. Wellings A.J., Keeffe D. and Tomlinson G.M., 1983, "A problem with Ada and Resource Allocation", Department of Computer Science report, University of York (also in Ada Letters January 1984)
13. Barton M.H., Skan P.L. and Aspinall D., 1984, "CHILL signals in a Distributed Environment", Proc. of Conf. on Hardware to support Distributed Languages, Bristol, UK
14. Needham R.M. and Herbert A.J., 1982, "The Cambridge Distributed Operating System", Addison-Wesley, London
15. Wirth N., 1982, Programming in Modula-2, Springer, Berlin
16. "An experimental Psychology Approach to Assessing the Comprehensibility of Alternative Language Features for Process Communication", 1984, Department of Computer Studies Report, Sheffield City Polytechnic
17. Blair G.S., Mariani J.A. and Shepherd W.D., 1983, Software Practice and Experience, 13, 45-58
18. Gouveia Lima I. et al., 1983, "Decentralised Control Flowed BASED on unIX, SIGPLAN Notices June 1983, 192-201
19. Shrivastava S.K. and Panzieri F., 1982, IEE Transactions on Computers, C-31, 692-697
20. Panzieri, F. and Randell, B., 1983, "Interfacing Unix to Data Communication Networks", Technical Report 190, University of Newcastle Computing Laboratory, Newcastle-upon-Tyne, UK
21. Bull G. and Lewis A., 1983, Software Practice and Experience, 13, 1075-1092
22. Hughes J.W. and Powell M.S., 1983, Software Practice and Experience, 12, 1099-1128
23. Bustard D.W. and Welsh J., 1979, Software Practice and Experience, 9, 947-957
24. Whitby-Strevens C. and Richards M. "BCPL: the language and its compiler", Cambridge University Press, UK
25. Ritchie DM et al, 1978, Bell System Technical Journal, 57, 1991-2019
26. Hoare C.A.R., 1978, Communications of ACM, 21, 666-677
27. Wirth N. and Jensen K., 1974, "Pascal: User manual and Report", Springer-Verlag, Berlin

28. Hughes J.W. and Powell M.S., 1984, "A strongly typed distributed virtual memory", in SERC Distributed Computer Systems, ed. D.A. Duce (see chapter 10)
29. Sloman M., Magee J. and Kramer J., 1984, "Building Flexible Distributed Systems in Conic", in SERC Distributed Computer Systems, ed. D.A. Duce (see chapter 6)
30. Buckley G.N. and Silberschatz A., 1983, ACM Transactions on Programming Languages and Systems, 5, 223-235
31. Bornat R., 1983, "Programming in Pascal-m", Computer Systems Laboratory, Queen Mary College, London, UK
32. Al-Mendhry F.R.A., 1982, "A distributed Pascal-Plus Implementation", Final year B.Sc project report, Department of Computation, UMIST
33. SERC, 1983, The Coordinated programme of research in Distributed Computing Systems, UK
34. Hoare C.A.R., 1974, Communications of ACM, 17, 549-557
35. Ritchie D.M. and Thompson K., 1978, Bell System Technical Journal, 57, 1905-1929

A strongly typed, distributed virtual memory

J. W. Hughes and M. S. Powell

Programming is predominantly the task of implementing data structures and the necessary operations required on them by the application. Within a single program, the task is well supported by the extensive data abstraction facilities of modern high level languages, but they make no such provision for data which is to outlive the execution of a program or to be shared among program executions. Such data is conventionally manipulated using a more primitive language of the operating system, filing system or data base. The work described here is an attempt to extend the data abstraction facilities of a modern high level language beyond the program boundary by strongly typing all data, both volatile and persistent, local and distributed, and consequently unifying the language by which it is manipulated.

5.1 INTRODUCTION

The concept of a filing system provided by a computer operating system emerged very early in the evolution of systems software, along with the concepts of FORTRAN as a high level programming language and the array as a data structure. All three provided convenient ways for the programmer to manipulate respectively the backing store, the instruction set and the memory whose hardware structures they closely reflect.

In the intervening years considerable advances have been made, particularly in the area of programming language design, which have resulted in the facility to express programs in problem-orientated rather than machine-orientated terms. These advances have centred around the concept of abstraction; program control structures have abstracted away from the machine jump instruction (13, 20), process and monitor structures have abstracted away from the sequential processor (2, 6, 19) and the concept of user defined types has abstracted away from the structure of memory (7, 14, 15, 20). Thus in the context of modern high level languages, files have become the Komodo dragon of programming, still reflecting more of the structure of the medium in which they are implemented than the structure of the information they store. Any long term structured data

has to be mapped onto the imposed file structures.

The work described here has arisen from examination of an existing information system for monitoring plant and men in a large press shop (10). Although the initial investigation was concerned with distributing this system, it soon became evident that a tool for constructing and manipulating the interrelated data structures needed would allow the designers and programmers of such a system to concentrate on the complexities inherent in the problem, whereas much of the complexity of the existing system lay in coping with the device dependent limitations of the filing system. Attempts elsewhere to date to overcome this problem have led to the emergence of data bases and database manipulation languages (4) independently of the important and significant advances in programming language design which have occurred.

Clearly there is no reason why the structure of the information which so-called files contain should be any different to the structure of the information manipulated by the programs that access and generate those files. If the conventional filing system is replaced by a structured data store, a natural and desirable consequence is that there is no need to design, implement and, more importantly, learn special purpose database description and manipulation languages. These are already provided conceptually by the type declaration and variable accessing facilities of most general purpose programming languages. Furthermore, within application programs, there is no longer a need for possibly complex and costly input and output routines which convert between the external file representation and the internal representation of the data types.

The replacement of a conventional filing system by a language specific structured data store is in line with the philosophy that the entire computer system should be language orientated. The flexibility of the UCSD (21) system is due largely to the fact that its operating system is designed specifically to support the implementation of Pascal programs and is itself written in Pascal. Similarly the potential of the PERQ derives from its underlying P-machine (22). A logical extension of this philosophy suggests that the data store (traditionally the filing system) should likewise be language orientated, in order both to simplify the programmers' task and to improve software portability. This paper describes such a Pascal orientated structured data store, its implementation and use. However, the concepts involved are applicable to any language employing user defined data structures. e.g. OBJ (4). The next section identifies the facilities required and the third describes the progress to date on their implementation.

5.2 FACILITIES REQUIRED

The system is required to support non-volatile abstract data structures in the spirit of the set theoretic types described by Hoare in (6) and ideally is to be built into a system orientated towards a modern high level language. By analogy with a filing system, the facilities required fall into two classes:-

i) Manipulation of the external data from within a program,

ii) Management, at the operator level, by means of a utility like a 'filer'.

If the system is bootstrapped, the latter can be implemented using the former i.e. the utility is written in the programming language around which the entire system is based, and may use its facilities for manipulating external structured data.

5.2.1 Types supported

In fact the high level language chosen, around which to build the system, was Pascal. The justifications for the choice are numerous - Pascal types closely approximate Hoare's abstract data types and the philosophy of building a machine and an operating system around a language has already been successfully put into practice for Pascal. Combining and extending these ideas to provide a Pascal based, structured, non-volatile data store is a logical development.

The set of predefined types and type constructors provided should be representative of abstract data types and compatible with those already provided in Pascal for program variables. The system should naturally support scalar types, both standard and user defined and the usual array and record constructors, including variants. The requirement for dynamic objects and recursive data structures was carefully considered and it was decided to follow Pascal and support both of these facilities via a pointer constructor. However, two specific recursive data structures have been provided to reduce the number of occasions on which the user will be reduced to using pointers directly. (See later). The real type, set and file constructors are not considered fundamental as they can be represented in terms of the other types. They have therefore been omitted from the current requirements list.

This then was the initial set of types, chosen as representative of abstract data structures in the Hoare sense (7):- standard and user defined scalars, pointers, records and arrays. Experience with using the system revealed the usefulness of the UCSD Pascal string and two frequently used dynamic structures which, as described below, are of particular use in defining and manipulating objects representing types. These are a variable length

sequence, implemented as a LIST and a MAP which represents a sparse function between types, implemented as a list of ordered pairs. These two constructors and strings have therefore been incorporated into the user interface to the data store at the operator level, although they were not part of the initial requirements and from a program they are still manipulated by the standard Pascal record and pointer mechanisms. They can be regarded effectively as mere short-hand notations in the user interface.

5.2.2 Manipulation from within a program

In traditional filing systems, the data representation within a file is generally different from that used to represent the same data within a program and is strongly influenced by the structure of the storage medium, rather than by the natural structure of the data itself. In these circumstances, considerable programming effort as well as machine time can go into providing the necessary transformations between representations whenever transfer of data between storage media occurs.

Ideally a structured data store should allow the same data representations to be used independently of the structure of their storage medium, just as programs written in a high level language are independent of the structure of the machine on which they are executed. Thus the normal data access mechanisms of the programming language should be available on all data objects independently of their location, as should assignment of values between structured types. The only new concept necessary is the way in which a program identifies an object external to it. This problem has already been solved for external files, but the present situation calls for stronger type checking between the program's and the data store's type definitions to be implemented. Providing a suitable virtual store can be implemented, the language extensions needed are minimal. In the existing system under discussion, the Pascal data accessing, assignment and dynamic store allocation mechanisms are implemented as a set of procedures (described in the third section) but it would be equally feasible to incorporate the virtual and external store concepts into a compiler and virtual machine.

5.2.3 Management of the Structured Data Store

In strongly typed languages, each data object has an associated type which is used by the compiler or the run time system to check the proper use of the data. In the case of external data, which outlives the execution of a program, it is necessary to maintain some representation of its type in order to provide the same level of security of proper use. Following this philosophy, the data store is strongly typed and all operations on it embody type checking. With each identifiable data object in the structured data store therefore are associated :-

i) A type which describes its structure

ii) Its value, of that type which may be manipulated, for example to create new values for existing or new data objects.

The data store management utility, in manipulating structured data objects, must therefore be capable of performing type checking and consequently the necessary type information must be stored as well as value information. Furthermore, the store management utility should provide the facility to create and update type information as well as value information. Thus, just as most filing systems embody the concept of a file directory which itself may be a file, so the structured data store may contain data objects whose values are type information. In those circumstances, the same utility can be used for the management of type information as is used for managing data values. In the one case information about the structure of type information is used to create or edit structured values representing type information, in the other user created type information is used for type checking during the creation or updating of data values. The use of the utility in the two cases is illustrated in figure 5.1. Its detailed use and precise user interface are described more fully in the third section.

5.2.4 Representations

In order for the user to make use of the type and value editors to interact with the structured data store, conventionally a textual interface is required, although any human sensory representation could be used. In either case, an alternative data representation to that used in the memory is necessary and the data manipulation software must be capable of performing the required transformations between alternative representations. In order to do so formatting information is associated with each type, which defines the textual (or alternative) representation of values of that type. Because types are defined hierarchically, the representation of a composite type is a function of the representations of its components and a formatting used to represent the constructor with which it is composed. The representations of standard types are standard and are specified in the standard type information. The representations of user defined types can be specified by using the type builder when the type itself is defined, or the representation can be changed by editing the formatting information in the type data structure. Thus the representation is type specific rather than 'persistent variable' specific. Although the facility is capable of being used for specifying general transformations of type representations, in the current system it has been used only for specifying two dimensional textual representations. The standard types have their usual textual representation with the added facility, similar to Pascal output parameters, to

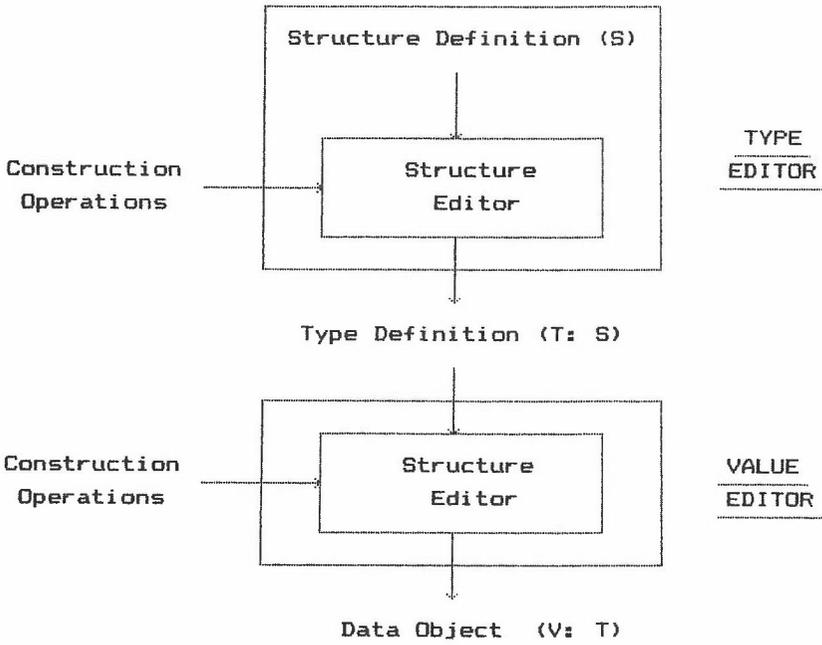


Fig.5.1

specify minimum field width and left, right or central justification within the field. The layout for array and record structures is specified in terms of prefix, infix and postfix strings which respectively precede, separate and follow the textual representations of the components of the structure.

5.2.5 Operations

The required facilities described above effectively extend the scope of Pascal structured variables beyond the scope of a Pascal program. More modern high level languages have extended the data abstraction concept beyond user defined types to incorporate the encapsulation of a data type together with operations on that type. The concept has been variously implemented as the class in Concurrent Pascal (2), the envelope in Pascal-plus (19) and the package in Ada (11). A logical further development of the permanent data concept incorporates operations into type descriptions. The concept of allowable operations upon data, or data components, of a user defined type includes formatting operations as a special case. As well as being used to provide type transformation operations, such operations could also be used to strengthen type checking by performing data validation of a non-syntactic nature and to provide higher level data base like query operations. Thus rather than programs defining the data which they manipulate, the data defines the operations by which it will be manipulated; each level in the hierarchy defining operations at the appropriate level.

5.2.6 Distributed Data

Only one external store has been considered above, however, the idea is readily extendible to identifiable external stores. This leads to the possibility of a physically or even geographically distributed data store where the user may require explicit control over the geographic location of data objects. Furthermore, the fact that values in the structured data store persist between program executions suggests that they may be shared between programs or users. In these circumstances, rather than being considered as classes, the data structures should behave more like monitors, providing exclusion on operations when appropriate. Previous work on nested monitor exclusion in distributed systems (17) suggests that not only is global exclusion a bottleneck, but local exclusion which makes no distinction between read and write access is also too strong. In order to be acceptably usable as a distributed data base, a structured data store of monitor-like data objects should provide exclusive writer and multiple reader access operations.

5.2.7 Requirements Summary

In this section, the concept of an external structured data store based on a strongly typed high level language has been developed. The facilities required on such a store are problem rather than device orientated. These include the normal data accessing facilities provided in most languages for accessing program variables, together with utilities analogous to filing operations which provide for the definition and representation of types and their values. Identification of multiple external stores leads to the concept of a distributed virtual store containing monitor like objects. The next sections describe progress to date on the implementation of such a store.

5.3 THE VIRTUAL MEMORY

The structured data store is implemented on top of virtual memory which may be distributed over a wide range of different memory devices. Conceptually the virtual memory implements a mapping between virtual addresses and the elements of the memory (words). Its interface, therefore, includes the following type definitions.

```
TYPE address = ...;
    word      = ...;
    memory kind = ...;
```

The memory kind enumerates the various memories over which the virtual memory is distributed. In an implementation which makes use of the processors main memory, a disk memory and a network link to other processors, each supporting the same kinds of memory, memory kind could be defined as below.

```
TYPE memory kind = (local memory, local disk, remote memory);
```

The address information required to access the virtual memory in this environment might be defined as follows.

```
TYPE address = RECORD
    CASE kind: memory kind OF
        local memory : (local : local address);
        local disk   : (disk : disk address);
        remote memory : (remote: remote address);
    END;

    local address = low address .. high address;
    disk address = RECORD
        volume name : volume identifier;
        block address : min block .. max block;
        word offset  : 0..max offset;
    END;

    remote address = RECORD
        node identifier : min node .. max node;
        internal address: address;
    END;
```

The address information for each particular memory

component can be chosen to suit the attributes of the device which implements it. Where a memory component contains further memory components, its address information may be defined recursively. This will generally be the case where a remote memory addressed via a network link is involved and allows, for example, memory accessing through network gateways.

The structure of a word can be chosen to suit the implementation environment. e.g. 1 bit, 1 byte, a 16 bit word etc.

The virtual memory interface must be able to support the following operations on the above types.

```
PROCEDURE allocate(VAR base: address; words: natural; where: memory kind);
PROCEDURE dispose(base: address; words: natural);
PROCEDURE read(location: address; offset: natural; VAR value: word);
PROCEDURE write(location: address; offset: natural; value: word);
```

Allocate and dispose support the initial creation and destruction of contiguous memory areas specified by their base addresses, sizes and the kinds of memory in which they reside. Read and write allow individual words to be accessed from any such contiguous memory areas. The virtual memory does not provide any operations to support the safe sharing of data objects between more than one user (mutual exclusion). Such operations are implemented at a higher level where advantage can be taken of the strongly typed nature of the structured data store (18).

Most of the internal structure of an address may remain hidden from users of the virtual memory except when the exact placement of a memory area is important. e.g. Long term data objects should be allocated on non-volatile memory devices such as disks and remote communication is only possible if both sender and receiver know where messages are stored.

The internal implementation of the virtual memory may use paging techniques to provide efficient access to words irrespective of the diverse structures of the physical memory devices over which it may be distributed.

5.3.1 Data Structure Accessing Operations

A set of standard structure accessing operations are build on top of the virtual memory interface. These include operations for creating, copying, disposing and accessing components of data structures of any of the supported types.

```

FUNCTION new structure(sort: type ref): address
PROCEDURE copy structure(source, sink: address; sort: type ref)
PROCEDURE dispose structure(old: address; sort: type ref)
FUNCTION scalar value(scalar: address; sort: type ref): ordinal value
FUNCTION index(array, index value: address; sort: type ref): address
FUNCTION field(record: address; field name: identifier; sort: type ref): address
FUNCTION dereference(pointer: address; sort: type ref): address
FUNCTION nil pointer(pointer: address; sort: type ref): Boolean;
FUNCTION string value(string: address; sort: type ref): string type
FUNCTION first list(list: address; sort: type ref): address
FUNCTION rest list(list: address; sort: type ref): address
FUNCTION empty list(list: address; sort: type ref): Boolean
FUNCTION first map(map: address; sort: type ref): address
FUNCTION rest map(map: address; sort: type ref): address
FUNCTION empty map(map: address; sort: type ref): Boolean
FUNCTION index map(map, domain value: address; sort: type ref): address

```

A complementary set of operations support the construction and interrogation of type, as opposed to value, information. The examples below show the operations which support the use of array type structures.

```

FUNCTION new array(index, element: type ref): type ref
FUNCTION index type(array sort: type ref): type ref
FUNCTION element type(array sort: type ref): type ref

```

Safe sharing of data objects between a number of users (mutual exclusion) may be achieved through the use of the acquire and release operations. In line with the above operations these must also specify the types of the objects on which they are to operate and the kind of exclusion which is required. e.g. read exclusion or write exclusion.

The operations on values are very similar to those provided in the instruction sets of high-level language machines, whether implemented in software or hardware. The primary difference lies in the fact that the operations of this system always have a detailed knowledge of the structure of the data objects on which they operate. This information is generally discovered by a compiler from a high-level language source text and limited amounts of it are encoded into the instructions which implement operations on the data objects it describes. However, the majority of the information is discarded before the data objects are ever created or used. Retaining all type information as an integral part of the system allows far more flexible use to be made of the values stored within it as the system knows so much more about the attributes of each object. Work reported in reference 16 indicates that the retention of type information may also have beneficial effects on efficiency.

In addition, where an operation refers to an address this may in turn refer to an object anywhere in the distributed virtual memory. For example copy structure can be used to move information of any type from one memory in a network to

another elsewhere in the network, or from a disk memory to the main memory of a processor, as easily as the information may be moved from one place to another within the same physical memory. This applies equally to information described by dynamic data types implemented using pointers. Copy structure will automatically transform the addresses which implement pointers in the source memory, into pointers suitable for use in the sink memory and create the appropriate values for them to refer to.

5.4 THE USER INTERFACE

Given the amount of type information which the structure store has about the objects stored within it, it is appropriate to allow the user access to the stored information via an interface which supports operations which are directly related to its structure. Nearly all conventional computer systems have interfaces which are unnaturally contorted away from the structure of the objects they manipulate due to a bias towards textual representations of data types, values and operations. e.g. Consider the amount of work performed by a compiler in extracting the essence of a program from its source text. If a program is represented directly as a data structure strongly related to the abstract syntax of the language in which it is 'written', all of conventional textual syntax analysis and much of conventional semantic analysis becomes unnecessary. In addition an executable representation of the program, its code, could be generated in an efficient manner from this data structure, or the structure could be executed interpretively.

The user interface to the structured data store is designed to allow direct manipulation of data objects in a fashion which is not biased towards textual representations but still allows the user to view information in a textual form. Many other non-textual formats are possible within the same framework. From this point of view, the facilities provided are similar to those found in program development environments which support structure editors such as Gandalf (5).

5.4.1 The Type Editor

Initially any object will have an undefined type and an undefined value. The first stage in the creation of a new object is therefore to define its type. This is done by using the type editor. Once the type of a new object has been defined the value editor may be used to give it a defined value. Both the type editor and the value editor are supported by a common structure editor which facilitates the stepwise creation and maintenance of arbitrary data structures.

The type editor allows the construction of type values of arbitrary complexity by using the facilities of the structure editor to edit values from some predefined domain

of types. In the current implementation the type value manipulated by the type editor is a mapping from type names to references to individual type values. The first entry in the mapping defines the type of the object to be created, subsequent entries define the types from which the new type is composed.

```

TYPE types      = MAP identifier TO type ref
  identifier    = string[identifier size]
  type ref     = ^type
  type kind    = (scalar, array, record, pointer, string, list, map)
  type        = RECORD
                name : identifier
                format: format information
                CASE kind: type kind OF
                  scalar: (scalar kind: scalar type)
                  array : (array  kind: array  type)
                  :
                  :
                  :
                END

```

An example will be used to illustrate how values of this type are manipulated by the type editor. The example is drawn from the area of computer aided teaching systems and involves the construction of a data object which is to represent a lesson. A lesson is to consist of some number of frames, each frame presenting a unit of information to the student. Frames will have distinguishing titles and contain references to succeeding frames which represent the continuation of the lesson. An example of the implementation of such a lesson is a 'Teach Yourself ...' book where frames are implemented by pages and continuations are implemented via page number references to succeeding material.

A type which describes the above lesson structure may be expressed textually, using the Pascal like type domain supported by the type editor, as shown below.

```

TYPE lesson    = MAP line TO frame ref
  frame ref   = ^frame
  frame      = RECORD
                title: line
                body : information
              END
  line       = string[line length]
  information = RECORD
                text      : LIST OF line
                continuation: lesson
              END

```

The data object must support operations which include the ability to add new frames to a lesson, the ability to define the information and continuations for such new frames and the

operations necessary to allow a student to 'browse' through the frames in the order indicated by the continuation information.

The lesson structure was defined textually above, however, the type editor's interface is not biased towards textual representations of type values but rather towards the manipulation of values from the domain of types which it supports. This makes the task of describing a non-textual interface textually rather difficult! An attempt will be made to overcome this problem by referring to the sequence of VDU screen images which the editor would present to the user during the course of the construction of the type lesson. Figure 5.2 shows the sequence of VDU screen images which are referred to in the paragraphs below.

The initial undefined state of an object's type is represented by an empty mapping. Figure 5.2a shows how the type editor displays this initial state. The top line on the screen always shows the type of the currently selected component of the value being displayed. In addition the currently selected component will be emphasised in some way, in this case by underlining. In this example the currently selected component is the entire types mapping which is empty. The editor adopts the convention that empty or undefined values are indicated by displaying their type names in pointed brackets.

The first step in defining the type of the object to be created is therefore to insert into the mapping a reference to the objects root type. In this case `lesson`. This is done by issuing the editors 'change' command. Commands may be given to the editor in the current implementation via single key strokes on a keyboard or by selecting command 'buttons' on the screen via a graphics tablet or light pen (these 'buttons' are not shown in figure 5.2).

When the change command is issued the display will change to that shown in figure 5.2b. The editor knows that the currently selected component is an empty mapping and therefore offers the possibility of inserting a new element into it. Selection of the insert option causes the display to change to that shown in figure 5.2c, as the editor enquires about the value of the element to be inserted into the map. In this case the element to be inserted is of a complex type which is best defined by defining its components individually. The user therefore enters an empty value and the display changes to that of figure 5.2d. At this stage the display is indicating that the types map now contains one element which is an undefined type ref.

The next stage is to create a type for the type ref to refer to. This is done by first selecting the type ref to be the currently selected component. This is achieved by 'pointing' at the type ref using a graphics tablet or light pen, or by using the arrow keys of a keyboard to position the screens cursor over it. When one of these actions is

A rectangular box representing a screen layout. At the top left, the text "<types>" is displayed. Below it, another "<types>" is shown, underlined. The rest of the box is empty.

(5.2a)

A rectangular box representing a screen layout. At the top left, the text "<types>" is displayed. Below it, another "<types>" is shown, underlined. At the bottom of the box, the text "[I]insert or [Q]uit?" is displayed.

(5.2b)

A rectangular box representing a screen layout. At the top left, the text "<types>" is displayed. Below it, another "<types>" is shown, underlined. At the bottom left of the box, the text "New value:" is displayed.

(5.2c)

A rectangular box representing a screen layout. At the top left, the text "<types>" is displayed. Below it, another "<types>" is shown, underlined. At the bottom left of the box, the text "<type ref>" is displayed.

(5.2d)

A rectangular box representing a screen layout. At the top left, the text "<type ref>" is displayed. Below it, another "<type ref>" is shown, underlined. The rest of the box is empty.

(5.2e)

Fig.5.2 Type Editor screen layout

```
<type>  
<identifier>  
<format>  
Type kind is scalar  
0  
0  
<id list>
```

(5.2f)

```
<identifier>  
<identifier>  
<format>  
Type kind is scalar  
0  
0  
<id list>
```

(5.2g)

```
<identifier>  
<identifier>  
<format>  
Type kind is scalar  
0  
0  
<id list>  
New value:
```

(5.2h)

```
<identifier>  
lesson  
<format>  
Type kind is scalar  
0  
0  
<id list>
```

(5.2i)

```
<type kind>  
lesson  
<format>  
Type kind is scalar  
0  
0  
<id list>
```

(5.2j)

Fig.5.2 Continued...

```
<type kind>  
lesson  
<format>  
Type kind is map  
MAP OF <type ref>
```

(5.2k)

```
<type ref>  
lesson  
<format>  
Type kind is map  
MAP OF <type ref>
```

(5.2l)

```
<type ref>  
lesson  
<format>  
Type kind is map  
MAP OF <type ref>  
  
New value: ^frame ref  
Not found. Shall I create it?
```

(5.2m)

```
<type ref>  
lesson  
<format>  
Type kind is map  
MAP OF <frame ref>
```

(5.2n)

```
<type ref>  
lesson  
frame ref
```

(5.2o)

Fig. 5.2 Continued...

performed the display will change as shown in figure 5.2e to indicate that although the whole types map is still displayed, that the first, and in this case only, element in it is the currently selected component.

If the editors 'view' command is used at this stage, a new type will be created and the previously undefined type ref will be made to point at it. In addition the display will change to show the current value of the new type instead of that of the types map. Figure 5.2f shows the new state of the display. The entire type value is the currently selected component and will be emphasised accordingly.

The new type has a default initial value. The first line <identifier> indicates that the name of the type is currently an empty string. This can be changed by selecting this string and using the editors change command to give it the value 'lesson'. The editor knows that this component of a type is a string and will accept a string of characters from the keyboard in response to the 'New value: ' prompt. Figures 5.2g, 5.2h and 5.2i show the various stages in defining the types name field.

The next field in the type is used to define its visual representation. It is currently undefined and will be left so for the purpose of this example. Figure 5.2j shows the display after the user has selected the type kind field. The default value of this is scalar and must be changed to map in order to define the type lesson. The fields which follow the type kind field are currently those which are associated with the scalar variant of a type (minimum ordinal value, maximum ordinal value and identifier list for enumerated types). Figure 5.2k shows the display after the change command has been used to change the type kind to map. Notice how the subsequent fields change to those associated with the map variant of a type.

In figure 5.2l the, currently undefined, range of the lesson map has been selected prior to giving it a value. The domain type is not displayed as it must always be the first component of the range type and is therefore deduced from the range of the map. For the purpose of this example the range type of the lesson map must be given a value which is a reference to the type frame. Figure 5.2m shows how the change command may be used to do this. In response to the 'New value:' prompt, the user types '^frame ref'. The '^' indicates to the type editor that the new value is to be searched for in 'appropriate' map type values in the structure being edited. An 'appropriate' map is one with the required range type, in this case type ref, and with a domain type which is consistent with the value following '^', in this case identifier. Maps in the structure being edited are searched starting with any that form part of the currently displayed value and then, if necessary, considering further maps by working back towards the root.

In the current example, the editor will report that it cannot find a type ref which points to a type named frame ref after searching the only appropriate map which is the types map at the root of the structure being edited. It therefore asks whether it should create a suitable type ref. Figure 5.2n shows what happens if the user tells it to do so. The editor creates a type value with its name field set to frame ref and adds a reference to it to the types map. Apart from the formatting information the type lesson is now complete.

Figure 5.2o shows the display after the user uses the editors 'return' command to return to the level in the structure being edited which is the immediate parent of the lesson type which has now been defined. This shows that the types map now contains two entries, one for lesson and one for its sub-type frame ref. The user can now select the frame ref entry and define it. This will lead to further entries being made in the types map which the user must define until the required structure is complete. In this way the editor supports the stepwise refinement of data types.

It is interesting to compare the number of key strokes required to enter the lesson type using this interface with the number required when using a conventional screen editor. The sequence of commands described above require 39 key strokes when using the editors keyboard interface only. There are about 42 characters in the corresponding part of the textual representation not counting redundant spaces or newlines.

```
lesson = MAP line TO frame ref
frame ref = ...
```

The number of key strokes is further reduced if the editor's graphics tablet or light pen interface is used but with the usual penalty of requiring the user to invest in a third arm.

The system has further obvious advantages over the use of conventional textual representations in that a data type, once completed, is ready for immediate use without any compilation process. Type checking is carried out incrementally during its construction.

5.4.2 The Value Editor

Types created by using the type editor are used to describe values to the value editor. The user interface to the value editor is identical to that used by the type editor because both utilities are built on top of a common structure editor. Figure 5.3 illustrates two views of a value of type lesson as seen through the value editor's interface. Figures 5.3a to 5.3c show the teacher's view during the initial stages of the construction of a new lesson. Figures 5.3d to 5.3f show the student's view during actual use of the teaching material constructed by the teacher. All of the displayed information is controlled by the format

```
<lesson>  
<lesson>
```

(5.3a)

```
<lesson>  
* <frame ref>
```

(5.3b)

```
<frame>  
<line>  
<text>  
<lesson>
```

(5.3c)

```
<frame>  
Pascal Identifiers  
  
A Pascal identifier starts  
with a letter which is  
followed by a sequence of  
letters or digits.  
  
* continuation ...
```

(5.3d)

```
<frame>  
continuation ...  
  
Which of the following is not  
a Pascal identifier?  
  
* Answer 1. mark1  
* Answer 2. markone  
* Answer 3. mark one
```

(5.3e)

```
<frame>  
Answer 2. markone  
  
Wrong!  
# is a letter and arnone is a  
sequence of letters.  
Review the definition.  
  
* Pascal Identifiers
```

(5.3f)

Fig.5.3 Value Editor screen layout

information created as part of the type value of a lesson. i.e. No further programming is required to implement this application above and beyond the initial construction of the type information.

In figure 5.3a the value editor has been called to edit an empty lesson. The first step in creating a new lesson is therefore to insert a new frame ref into it. Figure 5.3b shows the display after this has been done. The editor's 'view' command is then used and a new frame is created for the frame ref to refer to. The display changes to that shown in figure 5.3c and the user can go on to select and define the title, text and continuation of the frame. As the map type is used to define lessons, the structure of frames may be expressed hierarchically such that Pascal like scope rules apply to frame titles.

Figure 5.3d shows a frame as it might be seen by a student making use of the teaching material contained in a lesson on the subject of Pascal identifiers. The student is told that he may move through the lesson by selecting any of the starred entries and using the editor's 'view' command. The frame shown in figure 5.3d gives the student some information and invites him to move on to the next frame. When he does so the display changes to that shown in figure 5.3e and the student's understanding of the previous information is tested. Each of the continuations may lead to a new part of the lesson or take the student back to review earlier information. If the student selects 'Answer 2' in this case, the display changes to that shown in figure 5.3e, his mistake is explained and he is directed back to review the definition of a Pascal identifier.

Such lessons are only useful if they are kept on a non-volatile memory such as a disk. All of the operations described above can be performed without the lesson concerned being moved from the disk on which it is stored. The distributed nature of the data store would allow a lesson to be read by a number of students simultaneously. It could, of course, only be written by one teacher at a time. To prevent unauthorised changes being made to a lesson, a teacher might retain read exclusion after he had finished creating it, as exclusion is a non-volatile property of persistent objects within the data store.

5.5 CONCLUSION

Programming is, predominantly, the task of implementing data structures and the necessary operations required on them by the application. This process is well supported by most modern languages which include extensive facilities for data abstraction provided that the data to be manipulated is neither persistent nor distributed. Where either of these properties is required the programmer finds himself unable to abstract away from the storage representations of the abstract data structures he wishes to manipulate.

Because of this a very large class of conventional programs spend more time translating data structures from one memory representation to another than they do in performing the operations required by the application. e.g. A compiler generally spends more time performing lexical and syntactic analysis than it does in generating executable code. Both processes are necessary to translate from the textual representation of a program into a form which more closely describes the abstract structure of the program. One of the main reasons for using a textual representation for program data structures is that they are required to be persistent and the disk memories in which they are generally stored do not support abstract data structures directly.

Given an environment which supports persistent and distributed abstract data structures directly, many previously difficult programming problems become trivial. e.g. A program may be represented by a data structure which closely resembles the abstract syntax of the language in which it is 'written'. Such a data structure may be created by using the type editor to define its structure and visual representation. Programs may then be constructed and modified by using the value editor in much the same way as a syntax directed editor. If the facilities of the data store are extended to allow operations and consistency conditions to be associated with data types in much the same way that is possible with classes, then the value editor would also be able to perform semantic checking and execution of programs. This would be most useful if the operations and conditions associated with objects were defined using the same data structures used to represent programs.

A large program might be distributed between many programmers who could all work on their component of it, perhaps from separate workstations on a network, without the program becoming fragmented. Thus well defined interfaces between the components belonging to different programmers could be maintained at all times and incrementally checked during the construction of the program.

The structured data store described in this paper has been implemented under the UCSD (version IV) operating system and currently runs on a Sage IV computer. The user interface supported is a superset of that used in the examples in this paper and the underlying implementation permits data structures to be distributed over the main memory and disk memories connected to this machine. The mutual exclusion facilities described have not yet been implemented but work on this and the related problem of garbage collection are well under way. In addition a version of the system which allows data structures to be distributed around a network of memories belonging to four LSI-11/23s connected together by a Cambridge ring, has been implemented.

The system is quite compact. An early version ran on a 64kb Apple II and was only moved on to the Sage so that advantage could be taken of its large Winchester disk. The implementation is entirely written in Pascal and little attention has been paid to optimisation. However, the performance of the system as seen through its interactive interface is comparable with that of a good screen editor even though the current paging of the virtual memory only keeps one page at a time resident in the main memory.

Further work on the system will include the extension of the types provided to include class type objects so that user defined operations and consistency conditions can be made available. With these additions it should be possible to use the interface to the structured data store as a total replacement for conventional textually represented programming languages and operating systems. Alternative type domains, such as that of OBJ (4), will also be investigated using the facilities of the current system as a supporting tool. In addition the system will be used as a system development environment in which its ability to maintain parallel representations of system components in a hierarchical fashion will be utilised. Examples of such parallel components include requirements, specifications and implementations.

REFERENCES

1. M.P. Atkinson et al.
"An Approach to Persistent Programming"
Comp.J. 26, 4, pp360-365 (November 1983)
2. P. Brinch Hansen
"The Programming Language Concurrent Pascal"
IEEE Trans. on Software Engineering, 1, 2,
pp199-207 (1975)
3. C. Date
"An Introduction to Data Base Systems"
Addison Wesley, (1981)
4. A.N. Habermann and D. Notkin
"Gandalf Software Development Environment"
Carnegie Mellon University, Computer Science Dept.
(May 1982)
5. J. Goguen
"Parameterised Programming"
Proc. Workshop on Reusability in Programming,
Ed. J. Perlis (1983)
6. C.A.R. Hoare
"Monitors: An Operating System Structuring Concept"
CACM, 17, pp549-557, (1974)

7. C.A.R.Hoare
"Data Structures"
in Notes on Structured Programming, Dahl et al., (1972)
8. C.A.R.Hoare
"Communicating Sequential Processes"
CACM, 21, 8, pp666-677, (1978)
9. J.W.Hughes and M.S.Powell
"Program Specification Using DTL"
in Program Specification, Lecture Notes in Computer Science 134, Springer Verlag, (1981)
10. J.W.Hughes and M.S.Powell
"A Distributed Line-Monitoring System"
Cooperative Research Project, SERC/BL Systems Ltd.,
(1981-1983)
11. Ichbiah et al.
"Ada Reference Manual"
Castle House Publications, (1983)
12. M.A.Jackson
"Information Systems: Modelling, Sequencing and Transformations"
IEEE Proc. Int. Conf on Software Engineering, (1978)
13. D.E.Knuth and R.W.Floyd
"Notes on Avoiding "GO TO" Statements"
Inf. Proc. Letters, 1, pp23-31 (1971)
14. B.Liskov
"Modular Program Construction Using Abstractions"
in Abstract Software Specifications, LNCS 86, Springer Verlag, (1979)
15. D.Parnas
"Information Distribution Aspects of Design Methodology"
Inf. Proc. 71, 1, North Holland Pub. Co. (1972)
16. P.Schulthess and F.Vonaesch
"OPA - a New Architecture for Pascal-like Languages"
Comp. Arch. News 10, 6, pp9-20 (December 1982)
17. K.B.C.Tan
"An Adaptable Pascal Plus Virtual Machine Architecture"
M.Sc Thesis, UMIST, (1982)
18. S.T.Tye
"A General Virtual Machine for the Implementation of Concurrent High Level Languages"
M.Sc Thesis, U.M.I.S.T., (1984)
19. J.Welsh and D.W.Bustard

"Pascal Plus - Another Language for Modular Programming"
Software Practice and Experience, 9, (September 1980)

20.N.Wirth

"The Programming Language Pascal"
Acta Informatica, 1, pp 35 - 63, (1971)

21."UCSD Version IV User's Manual"

Softech Microsystems Inc. (1981)

22."Introduction to PERQ"

International Computers Ltd. (1982)

ACKNOWLEDGEMENTS

The work described was partially supported by SERC DCS research grant no. GR/B35062 and Cooperative award no. GR/B78861, in cooperation with BL Systems Ltd. The authors would also like to thank Chris Tan and Tony Tye for their suggestions and contributions to the work described.

Chapter 6

Building flexible distributed computing systems in Conic

M. Sloman, J. Magee, J. Kramer

6.1 FLEXIBILITY IN DISTRIBUTED SYSTEMS

Large computer systems are expected to have a long lifetime. However, they do not remain static during their operational life, but **evolve** as human needs change, the application environment changes and as new technology is incorporated. In fact the introduction of the computer system itself tends to act as a stimulus for change in the application environment, and so the services provided by the system must evolve.

In addition to **evolutionary change**, distributed systems must cater for **operational changes**. Components may have to be physically relocated in response to either personnel or installation changes. After failures of parts of the system, continued, possibly degraded, operation should be possible by manual or automatic reorganisation. Distributed systems should also cater for redimensioning: extension by addition of existing components or removal of superfluous ones.

A system must exhibit the property of **flexibility** in order to adapt to the above evolutionary and operational changes. The Conic approach to building distributed systems provides the capability for the system to evolve and change to meet changing requirements and conditions. A Conic distributed system (Kramer et al (1), Magee and Kramer (2)) can easily incorporate new functionality in response to evolutionary changes and allows reorganisation of existing components in response to operational changes.

We now refine and classify the flexibility requirements for distributed systems and then show how Conic meets these requirements.

Functional Flexibility - is the ability to modify a system to perform different or new functions. This can be achieved by the replacement of existing components or the addition of completely new ones. An important aspect of a system's functional flexibility is the ease with which one can identify the implications of a change ie. which other existing modules will be affected by the change.

Implementation Flexibility - allows for re-implementation

without a change in function. This could be for operational reasons, such as to improve performance, reduce running or maintenance costs, or increase reliability.

Topology Flexibility - the topology is the structure of the components (hardware, software) of the system. There are three aspects to be considered:

- i) **Physical** topology flexibility allows hardware components such as computers or transmission lines to be positioned or easily changed to meet the needs of the application. Typical changes would be to move a user's workstation from one office to another or to increase the number of workstations. There will always be fundamental limitations on the physical topology (eg.the maximum number of stations on a serial bus or the maximum distance between two stations on the bus). However, one should avoid making any artificial limitations such as system dependence on a specific network topology.
- ii) **Logical** topology flexibility in a system allows for any arbitrary communication patterns (between the software components) which meet the application requirements. The logical topology should be independent of the physical topology.
- iii) **Mapping** flexibility is needed for the mapping of software components and data onto the physical topology. Software components may be moved from one processor to another to optimise performance or to recover from failures.

Time Domain Flexibility - is an indication of when a system can be changed. A **static** system must be completely shut down in order to change any component. It may be necessary to wait for a time when it is quiescent such as during a maintenance period. **Dynamic** systems are flexible in that they allow functional, implementation or topology changes to a running system with no service interruption. In practice the disruption caused by a component change is dependent on whether it is being used by other components.

It has been widely recognised that in order to build large software systems, it is necessary to decompose the system into components which can be separately programmed, compiled and tested. The system is then constructed as a configuration of these software components. The separate activities of component programming and system building (configuration) have been referred to as "programming in the small" and "programming in the large" respectively (DeRemer and Kron (3)). In Conic this is reflected in separate component programming and configuration languages (1,2). In

this paper we will show how configuration management can be used to satisfy the above flexibility requirements.

In section 6.2 we explain how the Conic module programming language provides the necessary modularity characteristics for functional and implementation flexibility. The message primitives are described and it is shown that these meet the topology requirements because they employ indirect naming and provide transparency between local and remote communication. In section 6.3 we describe the Conic configuration language and physical topology of interconnected networks, which together meet the topology flexibility requirements. The facilities for structuring configurations to provide abstraction are also covered. In section 6.4 we explain the use of an on-line configuration manager to achieve dynamic configuration (time domain flexibility) in a Conic system. Finally we describe the current status of the Conic set of tools for building distributed systems.

6.2 CONIC MODULE PROGRAMMING LANGUAGE

6.2.1 Task Modules

Modularity is the key property for meeting the flexibility requirements. The Conic programming language is based on Pascal, which has been extended to support modularity and message passing primitives.

The language allows the definition of a **task module type** which is a self-contained, sequential task (process). A task module type is written and compiled independently from the particular configuration in which it will run ie. it provides **configuration independence** in that all references are to local objects and there is no direct naming of other modules or communication entities. This means there is no configuration information embedded in the programming language and so no recompilation is needed for configuration changes as is the case with other languages such as CSP (Hoare (4)) and ADA (5)).

At configuration time, **module instances** are created from these module types. Module instances exchange messages and perform a particular function in the system such as controlling a device or managing a resource. Multiple instances of a module type can be created on the same or different stations and a station can contain many different modules. This meets the requirements for mapping flexibility.

Conic modules have a **well defined interface** which specifies all the information required to use the module in a system. This is essential to provide implementation abstraction. The interconnections and information exchanged by modules is specified in terms of **ports**. An **exitport** denotes the interface at which message transactions can be initiated and specifies a local name and message type in place of the destination name and type. An **entryport** denotes the interface at which message transactions can be received and specifies a local name and type in place of the

source name and type. The binding of an exitport to an entryport is part of the configuration specification and cannot be performed within the task module programming language. Simple parameters (eg. integers, reals, booleans) may also be used at the interface to a task module type. Parameter values may then be passed to a module instance when it is created. This can be used to tailor a module type for a particular environment, for example to pass a device address to a device driver.

Figure 6.2.1 is an example of a simple bounded buffer task module.

```
TASK MODULE bound;
  ENTRYPORT putchar:char REPLY signaltype;
           getchar:signaltype REPLY char;
  CONST maxsize = 132;
  VAR inp,outp,content:integer;
      buf:ARRAY[1..maxsize] OF char;
  BEGIN
    inp:=1; outp:=1; content:=0;
  LOOP
    SELECT
      WHEN (content<maxsize)           {buffer not full}
      RECEIVE buf[inp] FROM putchar REPLY signal
      => inp:=(inp MOD maxsize)+1;
         content:=content+1;

    OR

      WHEN (content>0)                 {buffer not empty}
      RECEIVE signal FROM getchar REPLY buf[outp]
      => outp:=(outp MOD maxsize)+1;
         content:=content-1;
    END
  END
END.
```

Fig. 6.2.1 Bounded Buffer Task Module

There are two classes of ports which correspond to the message transactions classes described in 6.2.2. **Request-reply Ports** are bidirectional as they declare the types of values to be used for both a request message and the corresponding reply. **Notify Ports** are unidirectional ie. they have no reply part. For convenience, it is possible to define families (arrays) of identical ports. The following are examples of port definitions:

```
Exitport getch : char reply signaltype;
           alarm : boolean;
           datalinks [1..3] : message;
Entryport print : line reply status;
           message : msgtype;
           callsin [1..n] : printrequest reply printertype;
```

Ports define all the information required to use a module and so it is very simple to replace a module with a new or different version with the same operational inter-



Fig. 6.2.3 Request-Reply Transaction

6.2.2.3 Selective Receive - any of the receive, receive-reply, receive-forward, or receive-abort primitives can be combined in a select statement. This enables a task to wait on messages from any number of potential entryports. An optional guard can precede each receive to further define conditions upon which messages should be received. A timeout can be used to limit the time spent waiting in the select statement.

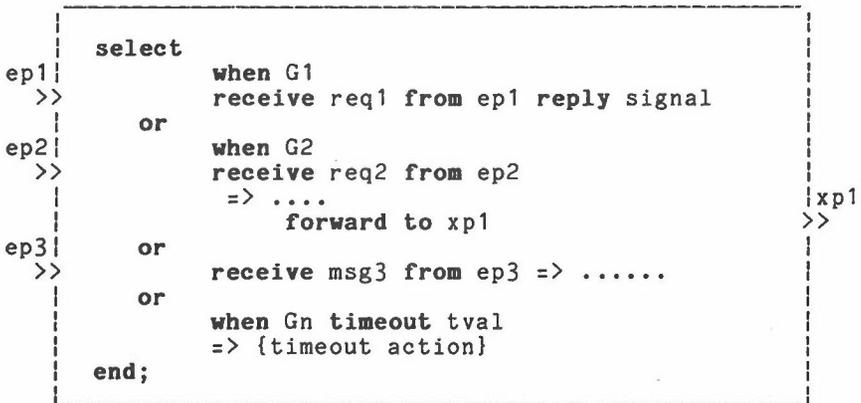


Fig. 6.2.4 Selective Receive

6.2.3 Input/output

The Conic Kernel provides a simple single primitive to support the implementation of device handlers as application tasks. The `waitio` procedure suspends a task until an interrupt occurs on the vector specified as a parameter eg. `waitio(100£8)`. This is similar to the facilities provided in Modula (Wirth (6)). The kernel does not convert interrupts into messages as in Ada (5) and SR (Andrews (7)), because such conversion complicates the kernel and increases response times to interrupts. If required, a simple task can convert an interrupt into a message eg. in order to queue interrupts. Task device drivers execute at the highest software scheduling priority level to ensure they are not pre-empted by other non-device handlers. When an interrupt occurs the scheduler is not called but rather the

hardware effectively schedules the relevant device driver task via the interrupt vector. Different device drivers may have different hardware priority levels, allowing nested interrupts. Figure 6.2.4 provides an example of a simple device driver which outputs characters to a Q-bus serial interface.

```
TASK MODULE serial_output( status,vector:natural) <system>;
{ 'status' is the address of the status register
  expressed as a 'natural' number.
  The task executes at 'system' priority i.e. with
  interrupts locked out.
}
ENTRYPORT output:char REPLY signaltype;

VAR xcsr:^natural;   {status register}
    xbuf:^char;      {data      "      }
    ch:char;

BEGIN
  REF(xcsr,status);   {REF converts 'natural' to }
  REF(xbuf,status+2); {a pointer type          }
  LOOP
    RECEIVE ch FROM output REPLY signal;
    xcsr^:=100#8;    {enable device}
    xbuf^:=ch;
    waitio(vector);
    xcsr^:=0;       {disable device}
  END
END.
```

Fig. 6.2.4 Device Driver Task Module.

6.3 CONIC CONFIGURATION LANGUAGE

One of the key elements in the provision of flexibility is the need to separate the programming of individual software components (task module types) from the building of a system from instances of modules. This has led to the development of the Conic configuration language which can be used to specify both the initial system and subsequent changes as described below. The following sections describe the essential properties which must be supported by a configuration language.

6.3.1 Context Definition

The context definition identifies the set of module types from which the system is constructed, and is provided by a `use` construct eg.

```
use bound, serial_output;
```

Modules communicate by typed messages so it is necessary for data type definitions to be shared between modules. Having to redefine types wherever they are used, would make type checking more difficult, more error prone

and require redundant effort by the programmer. Conic allows common datatypes and constants to be defined in separate **definitions** units. These are imported to make them accessible from both the programming and configuration languages eg:

```
from commstypes use datatype, acktype, buffer_size;
```

6.3.2 Instantiation

The **create** construct declares the named instances of module types to be created in the system. Instantiation parameters can be used to pass information such as device or interrupt vector addresses to device drivers. In order to cut down proliferation of names we allow the type identifier name to be overloaded and used as an instance name where only one instance of a type exists in a (sub)system.

```
Create serial_output (177560#8,100#8);
      datalink1,datalink2:CRdriver (retries);
```

Families (cf. arrays) of module instances can be created by specifying a 'range':

```
Create family k:[1..maxcalls]
      call [k] : call_handler (k);
```

6.3.3 Interconnection

The **link** construct specifies the interconnection of module instances by binding a module exitport to a module entryport. Both type and operation compatibility are checked so an exitport can only be linked to an entryport of the same data and transaction type. Multiple exitports can be linked to a single entryport which is particularly useful for connecting clients to servers (eg. a file server). A single notify exitport can be linked to multiple entryports which provides multidestination message transactions. Multidestination cannot be provided for request-reply ports because the semantics of dealing with multiple replies to a single request are unclear.

```
Link mod1.xp to mod2.ep;
      manager.errorout to logger1.errorin, operator.reports;
```

Families of modules and/or families of ports can be linked by defining range identifiers and associated ranges. This is merely a shorthand to save on repetitive link statements. The repetitions can be nested and are then performed in an analogous way to nested for-loops in Pascal.

```
Link family k:[1..maxcalls]
      call[k].in to manager.requests;
```

6.4.5 Mapping onto Physical Topology

The only constraint imposed by the configuration level on the interconnection of module exit and entryports is that they are compatible in terms of type and operation. The logical interconnection is completely independent of the physical configuration of the hardware components on which the system is to be run. The same logical configuration can be mapped onto a single computer, a closely coupled multi-processor station or distributed stations connected via an arbitrary network. It is thus important to separate the specification of a logical configuration from its mapping onto a physical configuration. Currently this mapping is performed by annotating a table produced by the configuration compiler. This maps the module instance names to physical stations.

The physical topology supported by a Conic system consists of Local Area Networks (LANs) interconnected by store-and-forward gateways (Sloman (8)). A station can communicate with any other station, if necessary via a gateway. This provides the required physical topology flexibility in that a variety of LANs can be used to suit a particular application requirements eg. Ethernet (9), Cambridge Ring (10), or one of the emerging IEEE LANs (11). The store-and-forward gateways provide the implementation flexibility at the data-link layer, allowing the interconnection of LANs of different transmission rates. The topology of interconnected subnets allows flexible extensions either of stations within a subnet or of subnets within an overall network.

6.3.5 Structuring Configuration Specifications

The modules in a distributed system often exhibit a hierarchical relationship. For example a database subsystem makes use of file servers, which may themselves consist of directory servers, record access servers and disc drivers. This structure can be represented by nesting software components at the configuration level by means of **group modules**. A group module type is a configuration specification and identifies a collection of module types, instances of those types and their interconnection. The constituent modules may be the primitive task modules containing a single process, described in section 6.2, or group modules. The group modules also have an interface defined in terms of exit- and entryports, as well as formal parameters. This structuring of the specification is essential for large systems with many module instances, otherwise the name space would become unmanageable and the configuration specification unreadable.

As mentioned, group modules provide configuration abstraction. The structure of a group module is defined by the **use**, **create**, and **link** constructs described previously. The interface to the module is also defined in terms of exit- and entryports and so from the outside it is not possible to distinguish between a task and a group module.

The group interface ports are bound to the ports of component module instances using link statements within the group module specification. These links are exitport-to-exitport or entryport-to-entryport eg.

```
Link grouentry to mod1.entry1;
    mod1.exit1 to grouexit;
```

This linking is merely a name mapping and does not entail any run-time overheads ie. there is no copying or queueing of messages at interface ports to group modules. The interface port name is global within the group specification and must be unique, whereas ports on different module instances can have the same name. The combined "module_name.port_name" must be unique within its scope (the group specification).

An example of a configuration description for a simple system which echoes characters at a terminal and uses the modules defined in the previous section is shown in Fig. 6.3.1. (Invert is a simple module which takes the front character from the bounded buffer and makes it available at its exitport outchar.)

```
GROUP MODULE buffer;
    ENTRYPORT inchar : char REPLY signaltype;
    EXITPORT outchar : char REPLY signaltype;

    USE bound,invert;
    CREATE bound;
        invert;
    LINK invert.getchar TO bound.getchar;
        inchar TO bound.putchar;
        invert.outchar TO outchar;
END.

GROUP MODULE echo;
    CONST status = 177560#8;
        vector = 100#8;
    USE serial_input,serial_output,buffer;
    CREATE Rx:serial_input(status,vector);
        Tx:serial_output(status+4,vector+4);
        B :buffer;
    LINK Rx.input TO B:inchar;
        B:outchar TO Tx.output;
END.
```

Fig 6.3.1 Configuration Description

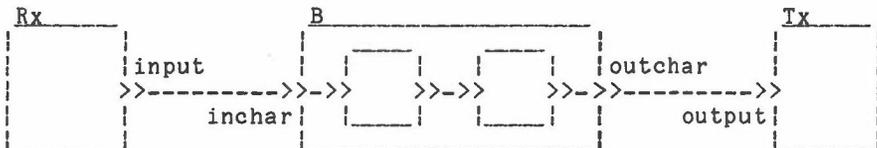


Fig 6.3.2 Configuration Diagram.

A **Segment Module** is a restricted form of group module in that the constituent module instances share an address space and so must be in a single station. The components of a segment module may share procedures and pass pointer values in messages. However there is no global data within a segment module. Any shared data must be encapsulated within a task module and references explicitly passed by messages. The only synchronisation primitives available for control of access to shared objects in a segment module are the message primitives. For example the Conic modules which implement the various layers of the communication system within a station pass pointers to message buffers in order to reduce the overheads of copying messages between these modules.

Also a group of tasks may be required to provide parallelism within a particular function. The terminal driver is an example of a set of closely related tasks which are grouped to form a segment module (ie. input from a terminal keyboard and output to its screen). This capability for parallelism at the task level encourages simpler cooperating sequential tasks rather than multi-threaded ones. The concept of a segment module is similar to that of a Guardian in Argus (Liskov (12)), but our segment modules do not automatically provide resiliency.

6.4 DYNAMIC CONFIGURATION

The configuration specifications described so far are essentially static. A Unix-based host development environment is used to produce load images which are down-line loaded into target distributed stations or put into ROM memory for embedded systems. In a Conic distributed system, where the operating and communication system is itself implemented as Conic modules, this static configuration is essential to provide the basic support for dynamic configuration in each station.

Dynamic configuration is necessary to provide the time domain flexibility mentioned in section 2. For many applications, it is too costly or unsafe to shut down a complete distributed system in order to change a component. A Conic system allows arbitrary, unpredicted modification and extensions to an existing system without rebuilding the entire system (2). It should be possible to perform incremental changes on the system without stopping the unaffected parts of the system. Changes which can be performed on a running system include:

- Installation and removal of module types;
- Creation and deletion of module instances;
- Changes to the interconnections between modules.

These changes are performed by submitting a change specification to an **on-line configuration manager** which validates the change and produces a new system specification incorporating the changes. It also generates the necessary commands to the operating system to perform the changes. Deleting a component would obviously affect other modules using it but changing the interconnections can often be

performed without affecting other modules.

6.4.1 Change Specifications

The change specification uses the configuration constructs described in section 6.3, but must also specify the inverse functions, namely:

unlink - disconnects a module exitport from an entryport.

delete - deletes the named module instances from the system. This can be performed only after all its ports have been unlinked.

remove - removes knowledge of the type from the configuration specification, and is valid only after all instances have been deleted.

For example, to remove the buffer from the example of Fig. 6.3.1 the change specification outlined in Fig. 6.4.1 would be submitted to the configuration manager.

```
CHANGE echo;
    UNLINK Rx.input FROM B.inchar;
        B.outchar FROM Tx.output;
    DELETE B;
    REMOVE buffer;
    LINK Rx.input TO Tx.output;
END.
```

Fig. 6.4.1 Change Specification

6.4.2 Configuration Manager

A change specification is submitted to a configuration manager which validates the specification, translates it into commands to the distributed operating system to execute the reconfiguration operations and produces the new system configuration specification. The configuration manager requires information on the current state of the system (eg. is a component type already in a station or will it have to be downloaded?) and must also have access to information necessary to perform validity checks. Some of this information can be obtained by querying the system to check its current state but type information is not maintained in stations and so must be held in an online database. A single change may result in a number of commands to the system (eg. create component instance => query resources, load type, instantiate).

The configuration manager currently being designed consists of three parts: a database describing the current system, specification translator and a command executor (Fig. 6.4.2). The initial version of the configuration manager will be centralised but later versions will be decentralised.

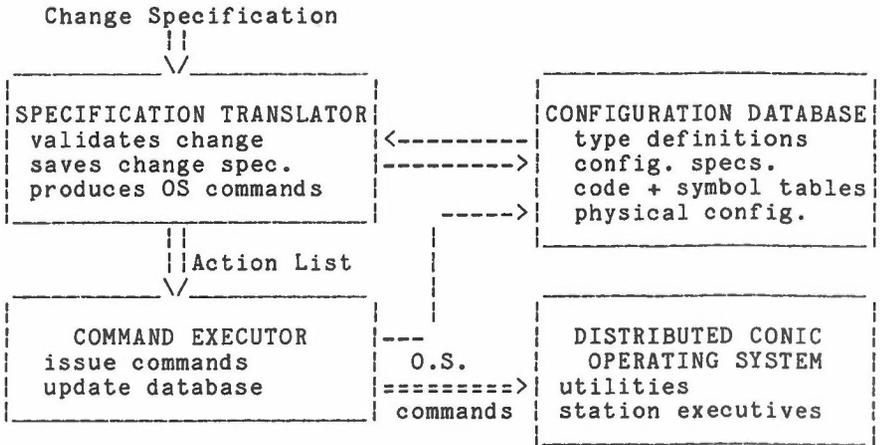


Fig. 6.4.2 Configuration Manager

6.4.2.1 Configuration Database. This holds the following information:

Task module types - the code generated by the module compiler, together with symbol tables for port names etc. These will be held in an intermediate machine independent target language called EM (Tanenbaum (13)) and the final target code generated when the type is loaded into a particular processor.

Type definitions - these are the definitions files of port and message types.

Configuration specification - the updated current specification of the system configuration together with a history of all change specifications in time order.

Physical system configuration - information on the subnets such as type, stations connected and current status. Also descriptions of the physical stations, including their resources such as memory and devices.

6.4.2.2 Specification Translator. This validates the change specification with respect to availability of resources (eg. memory or I/O devices) as well as type and operational compatibility for interconnections. The specification is translated into a sequence of simple commands to the operating system which are passed to the command executor. The translator uses the database to map the names in the specification into system addresses eg. a port address is specified by "subnet_id.station_id.module_id.port_id".

The change specification effectively produces a new system configuration but the change specification is kept in order to provide a history of changes. If required, changes can be reversed which is particularly useful for testing

The main influences on the design of the Conic distributed operating system (Magee (15)) were that the station executive should be small and efficient so that dynamic configuration could be provided on small microprocessor systems without backing store. This led to the principle of providing minimal functionality in the executive present in every station and rather implementing as much as possible remotely by utility modules. The executive should itself be configurable so that smaller ROM stations could omit the dynamic configuration support. This was accomplished by implementing most of the station operating system components as a set of Conic modules which can be configured using the static configuration facilities. The flexibility of the Conic module structure has been exploited in allowing distribution of the operating system components.

6.5.1 Station Executive

The executive is the set of Conic modules which together with the kernel manage the resources within a station and implement the communication primitives described earlier. The executive does **not** include device drivers which is usually the case in most operating systems. These are considered application utilities (see 6.5.2). Since flexibility is regarded as more important than performance the executive was mostly written as Conic modules, with the kernel being implemented in Pascal.

Station kernel - It is implemented in Pascal and provides multitasking and the primitives used by the executive's local management modules for task execution control and port linkage. It also provides the run-time support for the language extensions to Pascal ie. inter-task message communication within a station, timing primitives, and the simple interrupt mechanism described in section 6.2.3.

Communication system - This consists of a set of modules to support inter-station message passing. An exitport linked to a remote entryport is actually linked to a local communication module which formats a message by adding station addresses etc. and sends the message over the network to the remote station. At the remote station a communication module receives the message, strips off headers and then uses standard local Conic communication primitives to deliver the message. The communication system thus acts as a surrogate local source or destination for remote communication. The basic communication system provides a datagram service over a single subnet but configuration options include routing over interconnected subnets and a reliable virtual circuit service (8).

Local management - This is a set of four Conic modules: **modulemanager** deals with the loading of task types and creating instances; the **linkmanager** handles requests to link exitports of task instances within the station to either

local or remote entryports; **storeaccess** allows remote reading or writing of blocks of memory and is used for both down-line loading and remote debug; **errormanager** receives run-time error messages detected by the kernel or issued by a module reports them to a selected destination.

6.5.2 Utilities

The utilities provide the shared services traditionally found in operating systems. These utilities are implemented as normal Conic group modules and may themselves be distributed, but they are not found in every station.

File Server - This consists of a group of modules which perform the functions of file access (reading or writing blocks of a file), directory lookup (translates symbolic file names into file identifiers), management facilities (creating, deleting and renaming files) and disc drivers (reads or writes disc blocks).

Loader - This downline loads module type code into target stations. It obtains the type code from the file server and performs the final translation into the machine specific target code. It also relocates code to absolute memory addresses for processors with no memory management hardware. The memory instance address is obtained from the station's module manager.

The loader handles one code file at a time. However, more than one instance of the loader module can exist in a system and execute concurrently. The loader can of course be located on a different station from the file server.

Debugger - allows a remote module to be tested by via its message passing interface or by examining its memory space. The debugger provides the capability to construct test messages to send to a modules's entryports and to decode and display messages received from exitports. It can also the use store access manager of a (remote) station to read or write to the test module's memory space. The debugger operates on record structured objects, messages and variables. It obtains information on record structures from symbol table information stored on the file server.

It should be noted that neither the file server nor the console device module need be in the same station as the debugger module. Indeed, it is often useful to locate the console in the station being examined and the debugger in a remote station where more store is available.

Device handlers have been implemented for a number of network interfaces, terminals, discs etc. Our experience has been that these are comparatively simple to implement and integrate into the system. For example moving from the Omninet to Cambridge Ring required about two man weeks of programming effort for the new driver and requires a change of only a single line of a configuration specification.

6.5.3 Configuration Operations

The operating system utilities together with the local station executives, cooperate to implement the following dynamic configuration operations:

Load (stationid, codefile, moduletypeid)

The loader obtains the code size from the code file and sends a load request containing the moduletypeid to the target station. The station's module manger allocates memory space for code and returns the start address of the code segment. The loader forms a load image and sends load blocks to the station's storeaccess module.

Unload (stationid, moduletypeid)

The station's module manager deletes the moduletypeid and deallocates the storage for the type code. It can only be performed after all instances of the type have been deleted.

Create (stationid, moduletypeid, moduleinstanceid, parameterlist)

The station's module manager is given an identifier for the module instance and instantiation parameter values. The module manager assigns data segments, initialises control blocks etc. The module type code must have already been loaded into the station.

Delete (stationid, moduleinstanceid)

The module manager checks that the module ports are unlinked and deletes the module instance from the station.

Link (exitportid, entryportid)

The request to link an exitport to an entryport is sent to the linkmanager in the same station as the exitport. The entryportid is placed in the exitport's data structure (no informatin about a link is held at the entryport). A link to a remote entryport is actually made to the local communication system.

Unlink (exitportid, entryportid)

The entryport address is removed from the exitport data structure. If a request-reply transaction is in progress it will fail.

Start (stationid, moduleinstanceid)

The module manager requests the kernel to make the task module runnable.

Stop (stationid, moduleinstanceid)

The module manager in the target station requests the kernel to stop the task module.

Query (stationid)

This request to the module manager queries the state of the module instances in the station.

6.5.4 Performance

The approach adopted minimises the complexity of the station executive and instead moves all validation and the more difficult operations into the configuration management utilities. This has resulted in a configurable, efficient

operating system as indicated by the size and performance statistics for the prototype (LSI 11/23) system (given below). The figures are based on a station executive with a full set of local management modules, an Omninet data-link driver and 4 input message buffer modules, but no routing or virtual circuit service.

Station Executive Size (in kilobytes)

	CODE	DATA
Local management	5	2.4
Basic communications	2.5	2.5
Kernel	3.7	0.2

Total	11.2	5.1 K.bytes

Request Reply Performance (in milliseconds)

		LOCAL	REMOTE
0 byte request,	0 byte reply	1.5	19.2
100 byte request,	100 byte reply	2.2	21.5

Local Management Performance (in milliseconds)

Typical module creation time	5
Module deletion	50
Link	1.8
Unlink	2.1

The time to create a task module depends on the number of ports and the length of the module initialisation code. Module deletion takes much longer than creation since the executive checks that no exitports within the station are linked to the entryports of the module to be deleted.

6.6. CONCLUSIONS

6.6.1 Experience of Using Conic

A prototype system based on a network of LSI 11 micro-computers interconnected by an Omninet serial bus and Cambridge Ring has been in use at Imperial College for a number of years. We now have about 4 years experience of using earlier versions of the programming and configuration languages for implementing operating system utilities, device drivers, communication systems, and distributed simulations. It has been used both by experienced systems programmers and students for project work. The prototype software is also being used by the National Coal Board for implementing software for distributed underground monitoring and control stations, and by researchers at Sussex University for experimenting with adaptive control strategies.

This experience has shown that Conic provides an extremely simple yet very flexible approach to structuring a problem as a set of communicating components. Even comparatively naive student users have found Conic easy to

use for building both distributed and centralised concurrent systems.

6.6.2 Current Status

Reports defining the Programming (16) and Configuration (17) languages are available. A commercial product providing the set of tools for building distributed Conic systems, based on a Unix host environment and LSI 11/23 and 11/73 target microcomputers will be available by September 1984. Motorola 68000 based targets will be supported by December 1984. The task module compiler is based on the Amsterdam Compiler Kit (13) which has multiple back-ends and so simplifies the porting of Conic to new targets. The development tools allow static network configurations to be built and a prototype on-line configuration manager is being produced. The host system produces load images for stations which are down-line loaded over a network or can be placed in ROM.

Data-link drivers are available for the Cambridge Ring and Omninet Serial Bus. A distributed routing algorithm which caters for interconnected LANs and automatically updates routing tables to adapt to configuration changes has been implemented. There are a number of utilities such as terminal drivers, simple file servers and interactive debug tools.

6.6.3 Future Work

Fault Tolerant Conic. Some initial work has been done on incorporating fault tolerance techniques into a Conic distributed system (14). Both hot and cold standby redundancy can be supported. The configuration facilities are used to automatically switch to a **cold standby** module after a failure is detected. These can be used for applications which can accept the comparatively short time it takes to link and start a module. No state information is preserved. Applications which require completely transparent failure recovery can include a **hot-standby** module. The active module (performing the function) transfers state information at defined points during its operation to the passive 'hot-standby' module. In the case of a failure we automatically switch to the passive module and it assumes the active role. A new hot-standby passive module can be created. The hot standby approach to fault tolerance effectively masks module failures. This seems appropriate for many real-time applications. An interesting aspect is that the configuration manager can itself be made fault tolerant using these techniques.

Additional work is needed to incorporate the support for fault tolerance of transactions, such as the provision of atomicity (12).

Distribution of configuration manager. The prototype configuration manager will be implemented within the host development system and so will be centralised. We intend to

investigate alternative strategies for distributing the configuration manager both to improve reliability and to allow faster configuration changes.

Module Programming Languages. The Conic environment currently supports a single module programming language which simplifies some of the problems associated with transformation of information representation. The port data structures do not currently hold the type information needed for such transformations. We intend to investigate the problems associated with communication between both non-homogeneous computers and different languages.

The configuration flexibility provided by Conic could then be extended to building distributed systems consisting of modules implemented in other procedural languages such as Ada or even non-procedural languages such as Prolog. Conic could then provide the modularity framework for building distributed expert systems.

Module Behaviour. We are investigating the provision of specifications for the behaviour of individual task modules which could then be used in composition rules to specify the composite behavior of group modules. A sound, practical approach would provide the basis for module and system verification. It would allow analysis of a configuration specification for properties such as deadlock and whether it preserves specified constraints. Such specifications could also be used to predict the effect of configuration changes on the behaviour of a system.

ACKNOWLEDGEMENTS

We gratefully acknowledge the many useful discussions with our colleagues Kevin Twidle and Naranker Dulay, the support of the SERC under grant GR/C/31440 and the National Coal Board. The views expressed are those of the authors and not necessarily those of the NCB.

REFERENCES

1. J. Kramer, J. Magee, M. Sloman and A. Lister. CONIC: an integrated approach to distributed computer control systems. IEE Proc. Pt. E., 130:1, Jan. 1983, pp.1-10.
2. J. Magee and J. Kramer. Dynamic configuration for distributed real-time systems. Proc. Real-Time Systems Symposium, Arlington, Virginia, Dec. 1983, IEEE Computer Society, pp. 277-288.
3. F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. Proc. Conf. on Reliable Software, 1975. pp. 114-121.
4. C.A.R. Hoare. Communicating sequential processes. Comms. of the ACM, 21:8, Aug. 1978, pp.666-677.

5. USA Department of Defence. Reference Manual for the Ada Programming Language. Proposed Standard Document, July 1980.
6. N. Wirth. Modula: a language for modular multi-programming. Software Practice and Experience, 7, 1977, pp. 3-35.
7. G.R. Andrews. The Distributed Programming Language SR - Mechanisms, Design and Implementation. Software Practice and Experience, 12, 1982, pp. 719-753.
8. M. Sloman, J. Kramer, J. Magee, K. Twidle. A flexible communication system for distributed computer control. Proc. 5th IFAC Workshop on Distributed Computer Control Systems, May 1983, Pergamon Press.
9. XEROX Corporation. The ETHERNET: A local area network, data link and physical layer specifications. Version 1.0, September 1980.
10. Cambridge Ring 82 Interface Specifications, SERC, Sep. 1982.
11. IEEE Project 802 Local Area Network Standards, IEEE Computer Society, Dec. 1982.
12. B. Liskov and R Sheifler. Guardians and actions: linguistic support for robust distributed programs. ACM TOPLAS, 5:3, July 1983, pp. 381-404.
13. A. Tanenbaum, H. van Staveren, E. Keizer, J. Stevenson. A practical tool kit for making portable compilers. Comms. of the ACM. 26:9, Sep. 1983, pp. 654-662.
14. O. Loques-Filho. A fault tolerant distributed computer control system. Imperial College Ph.D. Thesis, Feb. 1984.
15. J. Magee. Provision of flexibility in distributed systems. Imperial College Ph.D. Thesis, April 1984.
16. J. Kramer, J. Magee, M. Sloman, K. Twidle. The Conic programming language: version 2.2. Imperial College Research Report, March 1984.
17. N. Dulay, J. Kramer, J. Magee, M. Sloman, K. Twidle. The Conic configuration language. Imperial College Research Report, March 1984.

The Cosy approach to distributed computing systems

P. E. Lauer

7.1 CONCURRENT, DISTRIBUTED AND SYNCHRONIZED SYSTEMS

Throughout this chapter we will be concerned with systems capable of concurrent behaviour. A system is concurrent if it is composed of several (sequential) subsystems whose respective behaviours may progress in parallel. A system is sequential if its behaviour can progress by an occurrence of only one event at a step, whereas the behaviour of a concurrent system can progress by occurrences of several events at a step. The notion of decomposition of a concurrent system into several subsystems introduces a notion of distribution in the sense that events must be assigned to subsystems in some particular way.

A concurrent system is synchronized if subsystems share events, and the interpretation of shared events is that the behaviours of the subsystems concerned may only progress by a coincident (simultaneous) occurrence of a shared event in all of the subsystems sharing the event. This implies that for a shared event to occur at some step, it must be capable of occurring at that step in all the subsystems sharing the event. It also implies that all behaviours of subsystems sharing an event will agree on the number and order of occurrences of that event.

Global behavioural properties of such systems include periodicity, absence of various types of deadlock, absence of starvation, mutual exclusion of sub-behaviours, etc. A system is periodic if all its behaviours are included in the multiples of some subset of behaviours called periods. The problem of analysis of the behaviour of a system can be greatly simplified if the system is periodic, since it can be reduced to the problem of analysis of the periods. A system is weakly deadlock-free, if and only if, for every behaviour there is at least one event of the system which may occur at the next step. A system is (strongly deadlock-free) adequate, if and only if, for every behaviour of the system and every event of the system there is a possible extension of the behaviour after which the event may occur. Notice that both types of absence of deadlock only say "may occur" and not "will occur". If the system in question is non-deterministic, then two events which may both occur at a

step, but whose occurrences exclude each other at that step, will lead to an arbitrary choice of one of the events to occur. At each recurrence of the step in the subsequent behaviour of the system the same choice could be repeated, leading to an infinite delay (starvation) of the event occurrence not chosen.

We will present a formal model, the COSY model, intended to support rigorously reasoned design, development and analysis of such concurrent systems.

7.2 DECISIONS INFLUENCING THE DESIGN OF THE COSY MODEL

To arrive at the level of abstraction of this particular model we made certain basic decisions:

1. We will concentrate on the concurrency, distribution and synchronisation structure of such systems based on the notion of primitive (uninterpreted) events. We will explicitly represent true concurrency (not interleaving), distribution and synchronisation throughout our model.
2. We will attempt to relate global behavioural (dynamic) properties of a system to structural (static) properties of its specification in some notation.
3. We will express the semantics of such specifications in terms of the notion of behaviour (history, trace) rather than in terms of more abstract notions such as the notion of function or the notion of relation. We will express the semantics of such specifications in terms of the notion of period as far as possible. Successful termination of some task the system may be said to be performing will be expressed not by the notion of halting, but by the notion of having completed some sub-period of a behaviour (which is analogous to having reached some "homing" state in a state oriented model).
4. The notation should be based on the notion of inherently periodic sequential subsystems and facilitate the construction of concurrent systems which are periodic in a more general sense.
5. The notation should allow maximum flexibility for decomposing a system into components to achieve greater concurrency of behaviour and/or differing distribution of events to components.
6. The notation and its formal semantics should permit its efficient implementation in a computer based environment for the design, development and analysis of concurrent systems. Although the verification of the correctness of the environment is based on rigorous mathematical reasoning, subsequent reasoning about systems developed by means of the environment should not require great

mathematical sophistication, but should be couched in a form comprehensible to non-mathematically specialized users.

7.3 TOWARDS THE COSY MODEL

Since we had decided to concentrate on concurrency, distribution and synchronisation structure and to express this structure explicitly, there was only one existing body of theory of concurrency which had been developed on the basis of similar decisions, namely General (Petri) Net Theory, see Brauer (1). This fact led us to adopt Net Theory as a standard semantics for our approach.

Within Net Theory systems are often specified by means of bi-chromatic directed graphs, called condition/event nets, involving two types of nodes representing states and transitions respectively, together with tokens whose movement over the graph indicates the asynchronous flow of control through the system. The notion of behaviour of such nets is made precise in Net Theory by the notion of partially ordered event occurrences and condition holdings, expressed as ordinary directed graphs called causal nets or occurrence nets. Thus we found in Net Theory much that suited our purposes.

However, there existed no non-graphic notation in which to express such condition/event nets and we felt that it is important to have such a linguistic notation if one wanted to support flexible and convenient transformation of one specification into another. This led us to develop the COSY notation as a linguistic alternative to condition/event nets. Such nets can be thought of as a generalisation of the notion of finite state automaton which naturally arises from the consideration of the fact that the finite state automata are closed with respect to composition in the sense of identifications of states, but are not closed with respect to composition in the sense of identification of transitions. This means that two finite state automata composed by identification of a state yield another finite state automaton, whereas if they are composed by identification of a transition the result is not a finite state automaton. On the other hand, both types of composition when applied to finite state automata considered as a special case of condition/event nets result in another condition/event net.

It is well known that the behaviour of a finite state automaton can be expressed by the language specified by some corresponding regular grammar, and hence it should be possible to express the behaviour of a condition/event net by the language specified by an appropriate generalization of regular grammars. Furthermore, since the structure of regular grammars corresponds well to the structure of their behaviours, it was to be hoped that the structure of their

generalized counterparts would also correspond well to the structure of their behaviours. Additionally, the use of language theoretic and automata theoretic methods is very prevalent throughout computer science in such areas as programming language design, compiler design, hardware design, network protocol design, etc. Hence, it was to be hoped that our generalization of these two types of methods would enjoy similar prevalence, and would not require any greater mathematical sophistication than that required for the application of more traditional language and automata theory.

The above considerations led us to begin our work by defining the semantics of the COSY notation by translation of a COSY specification into an equivalent labelled condition/event net and then utilizing the corresponding labelled causal nets to express their corresponding behaviours. However, we eventually abandoned this indirect method of defining the meaning of COSY specifications as the main semantics, in favour of a more direct method of defining an equivalent semantics in a language theoretic framework. This was due to a growing recognition of the fact that such a language orientation was more directly suited to the sum total of our decisions discussed above. We continued to use the net semantics of COSY whenever appropriate, and the equivalence of the two types of semantics allows transferral of results from net theory to the language oriented model and vice versa.

Finally, we rejected programming notations involving facilities for expressing concurrency, communication and/or synchronisation as a specification notation for two main reasons. First, we wanted to express "what" the behaviours of a system are, rather than "how" particular synchronisation facilities enforce these behaviours. Second, conventional programming language constructs themselves have synchronisation properties which are less important when they are used in sequential systems, but which assume great importance when they are used in concurrent systems. On the one hand such synchronisation properties encourage the designer to express part of the structure of the system in terms of the synchronisation primitives proper and express another part of the structure of the system in terms of properties of the more conventional programming language constructs. This may be an advantage from the standpoint of programming but may increase the difficulty of analysis of behaviour, since it tends to make the analysis of the synchronisation structure of a system dependent on the semantics of all the language constructs, rather than just on the semantics of the synchronisation primitives. Furthermore, to analyze the behaviours specified by a program one needs to abstract from "how" the program enforces the behaviour to "what" the behaviour is, so one might as well choose a specification notation sufficiently abstract to express behaviour directly. In that case the notation can also be used to determine, for example, whether two programs

implement the same system in the sense of enforcing the same abstract behaviour.

7.4 THE COSY MODEL

COSY (from Concurrent System) is a formalism intended to simplify the study of synchronic aspects of concurrent systems where possible by abstracting away from all aspects of systems except those which have to do with synchronisation.

A basic COSY path program, or generalised path is a collection of single paths enclosed in program and endprogram parentheses. A single path is a regular expression enclosed by path and end.

For instance:

```
PR = program
    P1: path a;b,c end
    P2: path (d;f)*;b end
endprogram
```

In every regular expression like the above, the semicolon denotes sequence (concatenation), and comma denotes mutually exclusive choice. The comma binds more strongly than semicolon, so that the expression "a;b,c" means "first a, then either b or c". An expression may be enclosed in conventional parentheses with Kleene star appended, as for instance "(d;f)*" which means that the enclosed expression may be executed zero or more times. The expression appearing between path and end is implicitly so enclosed, so a single path describes cyclic sequences of actions. The synchronisation among paths is due to common events ("b" in the above example). Every single path describes a sequential subsystem. The formal description of the COSY syntax may be found for instance in Lauer (2).

The semantics of generalised paths can be described by means of vectors of strings, an approach initiated in Shields (3).

With every single path $P = \text{path body end}$, we associate its set of events $Ev(P)$. In the case of example PR the events are:

$$Ev(P1) = \{a, b, c\}$$

$$Ev(P2) = \{b, d, f\}$$

which also indicates how events are distributed into subsystems.

For every regular expression E, let $|E|$ denote the regular language described by E. For every single path $P = \text{path body end}$ the language $|body|$ is called the set of cycles of P and denoted by $Cyc(P)$, i.e. $Cyc(P) = |body|$. For example PR we

obtain :

$$\begin{aligned} \text{Cyc}(P_1) &= \{a.b, a.c\} \\ \text{Cyc}(P_2) &= \{\{d.f\}*.b\} \end{aligned}$$

and they represent the periods of the inherently periodic sequential subsystems, namely the single paths.

From the set $\text{Cyc}(P)$ we construct the set of firing sequences of P , denoted by $\text{FS}(P)$, as follows:

$$\text{FS}(P) = \text{Pref}(\text{Cyc}(P)^*) = \text{Cyc}(P)^* . \text{Pref}(\text{Cyc}(P))$$

where for every alphabet A and every language $L \subseteq A^*$:

$$\text{Pref}(L) = \{x \mid (\exists y \in A^*) : x.y \in L\}.$$

The set $\text{FS}(P)$ is the set of sequences of event occurrences specified by the single path P . For example PR we obtain :

$$\begin{aligned} \text{FS}(P_1) &= \{a.b, a.c\}^* . \{e, a\} \\ \text{FS}(P_2) &= \{\{d.f\}*.b\}^* . \{e, d, \{d.f\}^*\} \end{aligned}$$

Consider a generalised path $P = \underline{\text{program}} P_1 \dots P_n \underline{\text{endprogram}}$ (or simply $P = P_1 \dots P_n$, where P_i 's are single paths. To model the non-sequential behaviour of $P = P_1 \dots P_n$, partial orders of occurrences of events will be constructed which are represented by vectors of strings.

A vector (x_1, \dots, x_n) is a possible behaviour of $P = P_1 \dots P_n$ if each x_i for $i=1, \dots, n$ is a possible firing sequence of P_i and furthermore, if the x_i 's agree about the number and order of occurrences of events they share. To formally define the set of possible behaviours or histories of P , vectors of strings are introduced together with a concatenation operation on them.

Let us consider the set $\text{Ev}(P_1)^* \times \dots \times \text{Ev}(P_n)^*$. If the vectors (x_1, \dots, x_n) and (y_1, \dots, y_n) belong to the above set their concatenation is defined as:

$$(x_1, \dots, x_n) . (y_1, \dots, y_n) = (x_1.y_1, \dots, x_n.y_n).$$

Let $\text{Ev}(P) = \text{Ev}(P_1) \cup \dots \cup \text{Ev}(P_n)$, and for $i=1, \dots, n$ let $h_i: \text{Ev}(P)^* \rightarrow \text{Ev}(P_i)^*$ be an erasing homomorphism given by:

$$(\forall a \in \text{Ev}(P)) : h_i(a) = \begin{cases} a & \text{if } a \in \text{Ev}(P_i) \\ e & \text{otherwise} \end{cases}$$

where "e" denotes the empty string.

Let $_ : \text{Ev}(P)^* \rightarrow \text{Ev}(P_1)^* \times \dots \times \text{Ev}(P_n)^*$ be the mapping defined as follows:

$$(\forall x \in \text{Ev}(P)^*) : x = (h_1(x), \dots, h_n(x)).$$

The set $\text{Vev}(P) = \{a \mid a \in \text{Ev}(P)\}$ is called the set of vector events of P. For example PR the vector events are :

$$\begin{aligned} \text{Vev}(PR) &= \{a, b, c, d, f\} \\ &= \{(a, e), (b, b), (c, e), (e, d), (e, f)\} \end{aligned}$$

or indicated by distribution into subsystems and "e" replaced by blank:

$$\begin{aligned} \text{Vev}(PR) &= P1: a.b.c. . . \\ &P2: .b. .d.f. \end{aligned}$$

Again the vector events indicate distribution of events to subsystems, and the sharing of events ("handshake" synchronisation) by sub-systems.

For $i=1, \dots, n$, let $[i]: \text{Ev}(P_1) \times \dots \times \text{Ev}(P_n) \rightarrow \text{Ev}(P_i)^*$ be a projection defined as:

$$[(x_1, \dots, x_i, \dots, x_n)]_i = x_i.$$

Note that: $(\forall x \in \text{Vev}(P)^*) (\forall i=1, \dots, n) : [x]_i = h_i(x).$

The set of all possible behaviours or histories of P, the vector firing sequences of P, denoted by $\text{VFS}(P)$, is defined by:

$$\text{VFS}(P) = (\text{FS}(P_1) \times \dots \times \text{FS}(P_n)) \cap \text{Vev}(P)^*.$$

The set $\text{FS}(P_1) \times \dots \times \text{FS}(P_n)$ in the definition of $\text{VFS}(P)$ guarantees that each string component x_i of a history $x = (x_1, \dots, x_n) \in \text{VFS}(P)$ is a firing sequence of the path P_i , and the set $\text{Vev}(P)^*$ guarantees that all these firing sequences agree about the number and order of occurrences of events they share.

The set $\text{VFS}(P)$ can be treated as a formal description of the execution semantics: "execute as possible (i.e. not necessarily maximally concurrent)".

If we define generalised periods by :

$$\text{Per}(P) = \{x \in \text{VFS}(P) - \{e\} \mid \forall i \{1, \dots, n\} : [x]_i \in \text{Cyc}(P_i) \vee [x]_i = e\}$$

then P is periodic if

$$\text{VFS}(P) = \text{Pref}(\text{Per}(P)^*) = \text{Per}(P)^* \cdot \text{Pref}(\text{Per}(P))$$

For example PR which is periodic this gives:

$$\text{Per}(PR) = \{a.c, a.b, d.f.a.b\} \quad \text{and}$$

$$\text{VFS}(PR) = \{a.c, a.b, d.f.a.b\}^* \cdot \{e, a, d, d.f, d.a, d.f.a\}$$

and showing the distribution into subsystems in $\text{Per}(\text{PR})$ we get:

P1: a.c a.b e.e.a.b
 P2: e.e , e.b , d.f.e.b

Let $\text{ind} \subseteq \text{Ev}(P) \times \text{Ev}(P)$ be the following relation:

$$(\forall a, b \in \text{Ev}(P)) : (a, b) \in \text{ind} \iff (\forall \text{Pi}) a \notin \text{Ev}(\text{Pi}) \text{ or } b \notin \text{Ev}(\text{Pi}).$$

The relation ind is called the independency relation. Note that:

$$(a, b) \in \text{ind} \iff a \neq b \ \& \ \underline{a.b} = \underline{b.a} \iff (\forall i) [\underline{a}]i \neq e \implies [\underline{b}]i = e.$$

The definition of ind implies that only independent events may occur concurrently. However, independent events may not always occur concurrently or may never occur concurrently at all.

Before we give formal definitions of the notions of sequential, concurrent and maximal concurrent reachability we will reconsider the behaviour of example PR. At the beginning events a and d may occur in one step or one after the other in two steps. After their occurrence in one step both c and f may occur in one step or separately. After the occurrence of c and f in one step again a and d may occur in one step, and so on. On the other hand, a behaviour could start with an occurrence of a in one step, after which b and d may occur. Hence b may occur, a case that does not arise when independent events always occur in one step.

This example shows that the semantics "execute as much as possible in parallel" may not be equivalent to the semantics "execute as possible". Let us analyse this problem formally. First of all, we must formally define the semantics "execute as much as possible in parallel".

Let $P = P_1 \dots P_n$ be a generalised path, and let $\text{Ind}(P) \subseteq 2^P$ be the following family of sets of operations:

$$\text{AGInd}(P) : \iff (\forall a, b \in A) (a, b) \in \text{ind}.$$

In other words elements of $\text{Ind}(P)$ are sets of independent events. If $A = \{a_1, \dots, a_k\} \in \text{Ind}(P)$, then $\underline{a_1 \dots a_k} = \underline{a_{i_1} \dots a_{i_k}}$ for any permutation i_1, \dots, i_k so we may write $\underline{A} = \underline{a_1 \dots a_k}$.

For every $x \in \text{GVFS}(P)$, an event $a \in \text{Ev}(P)$ is said to be enabled at x if and only if:

$$(\forall i = 1, \dots, n) a \in \text{Ev}(P_i) \implies [x]i.a \in \text{GVFS}(P_i).$$

For every $x \in \text{GVFS}(P)$, a set of independent events $\text{AGInd}(P)$ is said to be concurrently enabled at x if and only if every $a \in A$ is enabled at x . For every $x \in \text{GVFS}(P)$, let $\text{enabled}(x)$ denote the family of all concurrently enabled sets of events at x .

A concurrently enabled set at x , $A_{Genabled}(x)$, is said to be maximally concurrent if and only if it may not be extended, i.e. iff $(\forall B_{Genabled}(x)) A \subseteq B \implies A=B$.

For every $x \in VFS(P)$, let $maxenabled(x)$ denote the family of all maximally concurrent sets enabled at x . Of course $maxenabled(x) \subseteq enabled(x)$.

Let $-s>$, $-c>$, $-m>$ $\subseteq Vev(P)^* \times Vev(P)^*$ be the following relations:

$$\underline{x} -s> \underline{y} : \iff (\exists a \in Ev(P)) \{a\}_{Genabled}(\underline{x}) \ \& \ \underline{y} = \underline{x}.a,$$

$$\underline{x} -c> \underline{y} : \iff (\exists A_{Genabled}(\underline{x})) \underline{y} = \underline{x}.A,$$

$$\underline{x} -m> \underline{y} : \iff (\exists A_{Gmaxenabled}(\underline{x})) \underline{y} = \underline{x}.A.$$

The relations $-s>$, $-c>$, $-m>$ are called respectively: the sequential reachability in one step, the concurrent reachability in one step, and the maximally concurrent reachability in one step.

$$VFS(P) = \{\underline{x} | \underline{e} -s>^* \underline{x}\} = \{\underline{x} | \underline{e} -c>^* \underline{x}\}.$$

The above fact states that VFS is fully characterisable by the relation $-c>$. The computer based environment, BCS, to be discussed in a later section is nothing but an implementation of the relation $-c>$, (2) and Hamshere (4).

The relation $-m>$ is that mathematical object which represents the maximally concurrent evolution, i.e. one step under the rules of the semantics "execute as much as possible in parallel".

$$\text{Let us define: } VMFS(P) = \{\underline{x} | \underline{e} -m>^* \underline{x}\}.$$

The set $VMFS(P)$ represents all histories that may be reached by a maximally concurrent evolution of the system (the vector maximal firing sequences), so it may be treated as a formal description of the execution semantics: "execute as much as possible in parallel".

$$\text{For every } X \subseteq Vev(P)^*, \text{ let } Pref(X) = \{\underline{x} | (\exists \underline{y} \in Vev(P)^*) \underline{x}. \underline{y} \in X\}.$$

We will say that P is completely characterised by maximally concurrent evolution if and only if:

$$VFS(P) = Pref(VMFS(P)).$$

The above equality is a formal expression of the fact that the semantics "execute as much as possible in parallel" and the semantics "execute as possible" are equivalent. In Janicki et al (5) we give some sufficient conditions for a system to be completely characterized by maximal concurrent evolution alone, which permit the time taken to simulate behaviours of the system to be improved by a factor of eight.

The formal model of behaviour permits us to speak formally of dynamic properties of systems specified by a generalised path $P=P_1...P_n$.

We say that $P=P_1...P_n$ is deadlock free if and only if:

$$(\forall x \in VFS(P)) (\exists a \in Ev(P)) x.a \in VFS(P),$$

that is every history x may be continued.

We say that $P=P_1...P_n$ is adequate if and only if:

$$(\forall x \in VFS(P)) (\forall a \in Ev(P)) (\exists y \in Ev(P)^*) x.y.a \in VFS(P),$$

that is, if every history x of P may be continued eventually enabling every event in P . Adequacy is a property akin to absence of partial system deadlock. More details can be found in (3), Shields (6) and Lauer et al (7).

7.5 FORMAL RESULTS ABOUT THE MODEL

A large number of formal results have been proved about the COSY model in the past ten years. We briefly discuss three types of such results below.

7.5.1 Static Criteria for Adequacy and Periodicity

We have discovered a number of sub-classes of COSY specifications which permit the deduction of their adequacy from the structure of their description. For larger sub-classes we have obtained results which permit the compile-time calculation of bounds for the runtime required to check adequacy dynamically. Details can be found in (3), (6), (7) and Best (8).

7.5.2 Equivalence of COSY Specifications

On the basis of the relationship between partial orders and vector firing sequences we have introduced a notion of equivalence for COSY specifications with different numbers of sequential components, and we have discovered two normal forms for vectors of firing sequences, see Janicki (9).

7.5.3 Mathematical Results about Vector Firing Sequences

Shields (6) and (10) are extensive presentations of the mathematical properties of vector firing sequences, and in particular the notions of periodicity, and various order theoretic (e.g. existence of least upper bounds) properties are subjected to a careful study.

7.6 TRANSFORMATIONAL DEVELOPMENT OF SPECIFICATIONS

By transformational development of a specification we mean the development of a specification starting from some initial specification by a sequence of syntactic transformations which produce intermediate specifications, until some specification is reached which is considered final for the purposes for which it was required. In the COSY theory we have developed two kinds of techniques for transformational development which guarantee that semantics of some kind are preserved in each step of the development process.

7.6.1 Constrained Expansion and Reduction Rules

In the first type of technique, substitution, expansion and transformation rules are used to generate specifications which are adequate by construction. It can be shown that all specifications which have been generated in this way have a special behaviour, they are all periodic. Reduction and excision rules can be used to decrease the complexity of a specification or to break it into several disjoint specifications. Details can be found in (3), (10) and Hillen (11).

7.6.2 Decomposition of Sequential into Concurrent Systems

The second type of technique involves the decomposition of sequential systems into semantically equivalent concurrent systems. Given a sequential COSY specification one can define abstract resources which the system is to use and there exists an algorithm for decomposing the sequential specification into a maximally concurrent and functionally equivalent one.

This is important because of the following theorem and others proven in (9) and Janicki (12).

Theorem

Given sequential and concurrent specifications S and R , respectively:

If C and S are functionally equivalent then C is adequate.

This means that it is possible to verify the adequacy of COSY specifications by first giving a sequential solution which is adequate by definition, transforming it into a concurrent specification and using the above mentioned algorithm to verify functional equivalence, which by the above theorem guarantees the adequacy of the concurrent specification.

7.7 DEVELOPMENT OF A COMPUTER BASED ENVIRONMENT FOR COSY

We have implemented a computer based environment which supports the systematic design, development and semantic analysis of specifications written in the COSY notation.

7.7.1 The Basic COSY System : BCS

This system permits one to enter basic COSY specifications via a context editor or a screen editor, to incrementally debug and compile such specifications, and to simulate them interactively or automatically. The system includes an automatic logging system, a version management system, a facility for producing reports during interactive use of the system, flexible context switching between different sub-facilities of BCS and between BCS and the host operating system without losing contexts switched from, and many other facilities which support the general software development process. Lauer (13) is a detailed user introduction to COSY and BCS.

7.7.2 The COSY Dossier

The dossier concept was introduced to integrate the facilities belonging to BCS, for organizing the information obtained about systems by means of BCS, and for managing the process of system development and analysis in general. Details can be found in (2).

7.8 HIGH LEVEL COSY NOTATION

During the development of the COSY notation the need to provide the system designer with a tractable tool for design became apparent. Path expressions do not do this very well in themselves. Thus, one effort of our work has been towards the development of higher level descriptive notations. Two more notations were developed:

Macro notation To formulate COSY systems in a concise and general way, generators have been introduced in the notation which may represent a finite but possibly indefinite number of repetitions of regularities of structure in basic COSY programs, see Lauer et al (14) and Cotronis (15).

System notation To express hierarchy, modularity, and levels of abstractness of a design, the notation has been equipped with a (SIMULA) class-like macro construct for which we use the term "system" (14) and Torrigiani and Lauer (16).

The semantics of programs in the macro and system notations

can be given in terms of the semantics of equivalent basic programs generated by expansion. A computer based environment called CS is being implemented which extends the BCS environment to include the high level notations, see Lauer (17).

7.9 APPLICATIONS OF COSY

7.9.1 Operating System Problems

The COSY model has been applied in the analysis of most of the standard synchronisation problems and solutions discussed in the literature on operating systems. These solutions have been verified within the COSY model and published in the papers (2),(3), Lauer and Shields (18) and (19), and Lauer et al (20).

7.9.2 Network Protocols

COSY has also been used to specify and verify network protocols and has been found to nicely extend the finite state machine based verification techniques prevalent in this area. For details see Cotronis and Lauer (21).

7.9.3 Train Journeys

COSY has also been applied to study the behaviour of systems other than computer systems. One such application is in the specification of the behaviour of a highly concurrent model train set. Details are in Shields (22), Janicki and Lauer (23), Devillers (24) and Koutny (25).

7.10 VLSI IMPLEMENTATION OF COSY

Recently we have developed a VLSI implementation of COSY by implementing each sequential subsystem by a PLA, achieving concurrency by running the PLAs simultaneously, and enforcing synchronisation by a decentralised "busy waiting" mechanism connecting these PLAs, see Li and Lauer (26).

7.11 RELATION OF COSY MODEL TO OTHER APPROACHES

The COSY model has been formally related to a number of differing approaches to distributed systems. One group of workers has tended to concentrate on the development of mathematical models of such systems. Another group has concentrated more on the development of programming notation for such systems. We will discuss these two groups separately below.

7.11.1 Other semantic models

7.11.1.1 Translation of COSY to Labeled Petri Nets : We will briefly explain the translation from basic COSY programs to labelled marked condition/event nets. The current net semantics of basic COSY programs is obtained by translating each component sequential path into a labelled state machine represented as a net, i.e. representing transitions by boxes. For example, the paths :

P1: path a;b;a end P2: path a,c;d end

would individually give rise to :

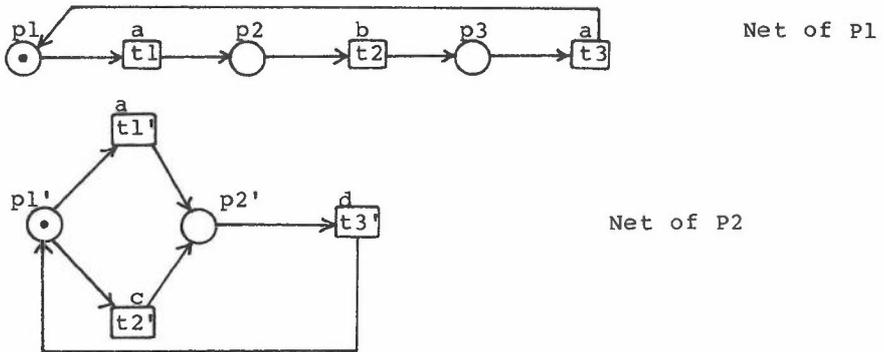


Fig.7.1 Nets of individual paths

Once the nets corresponding to the individual paths have been obtained, for example, two nets called N1 and N2, one applies a composition rule denoted by " \otimes " to the two nets, written $N1 \otimes N2$, constructed from N1 and N2 by the identification of transitions with the same label.

We may now give the construction of $N1 \otimes N2$ from nets N1 and N2, and illustrate it with the two example paths above :

1. The set of places of $N1 \otimes N2$ is the set theoretic union of the sets of places of N1 and N2, with inherited markings.
2. Suppose t is a transition in either N1 or N2 such that no transition in the other net is labelled with the label of t, then $N1 \otimes N2$ contains a transition \underline{t} , with the same label as t, whose input and output places are the same as those of t (recall 1).
3. Suppose t1 and t2 are transitions of N1 and N2, respectively, with the same label, then $N1 \otimes N2$ contains a transition $\underline{(t1, t2)}$ with the same label as t1 and t2 and whose set of input (respectively output) places is the union of the sets of input (respectively output) places

of t_1 and t_2 .

" \otimes " may be shown to be commutative and associative. If $P = P_1 \dots P_m$ and N_i is the marked labeled state machine associated with P_i , then the net associated with P is defined to be $N_1 \otimes \dots \otimes N_m$.

The result of applying these rules to our example paths P_1 and P_2 is then :

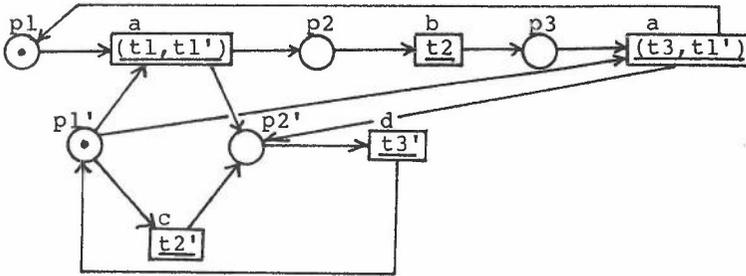


Fig.7.2 Net of $P_1 \otimes$ Net of P_2

A number of results about the correspondence of COSY sub-languages and sub-classes in the hierarchy of Petri Nets classified with respect to structural (and related behavioural) complexity, have been obtained. This formal correspondence permits the transferral of results from one model to the other. Details are in (7) and (8).

7.11.1.2 Calculus of Communicating Systems : Shields (10) contains some initial comparisons of VFS semantics and Milner's calculus for communicating systems CCS, see Milner (27).

7.11.1.3 Structural Operational Semantics : Li and Lauer (28) presents a semantic for COSY notation in terms of the structural operational approach of Plotkin (29), and demonstrates that this semantics is equivalent to the VFS semantics for COSY. Previous definitions of concurrency in the operational approach reduced concurrency to interleaving, whereas our definition models true concurrency. The operational approach has been used to define the semantics of a number of programming languages including ADA, Edison, CSP and CCS, see Li (30), and our result allows a straight forward extension of these semantic definitions to model true concurrency.

7.11.2 Programming Notations

The COSY model has been used to specify and study behavioural semantics of a number of programming notations involving synchronisation or communication primitives.

7.11.2.1 Communicating Sequential Processes : Lauer (31) relates the COSY model to Hoare's communicating sequential process notation and its corresponding trace semantics, see Hoare (32).

7.11.2.2 Extended Semaphore Primitives : Shields and Lauer (33) contains a formal semantics of Agerwalla's extended semaphore primitives (34) and develops a concurrency preserving translation from programs involving these primitives into the COSY notation.

7.11.2.3 Monitors : Cotronis and Lauer (35) shows how various types of monitors can be formulated in the high level COSY notation as system definitions. This allows one to treat such monitors as high level primitives in COSY, and illustrates the use of COSY as a software specification tool.

7.12 ACKNOWLEDGEMENTS

The work outlined here was financed by a number of grants from the Science and Engineering Council of Great Britain. Contributors to the work on the COSY model are indicated by the appearance of their names in the references.

7.13 REFERENCES

1. Brauer, W. (Ed.), 1980, 'Proceedings of the Advanced Course on General Net Theory of Processes and Systems', Hamburg, 1979, Lecture Notes in Computer Science 84, Springer Verlag.
2. Lauer, P.E., 1982, 'Computer System Dossiers', Distributed Computing Systems, Academic Press Inc., 109-147.
3. Shields, M.W., 1979, 'Adequate Path Expressions', Proceedings of the International Symposium on the Semantics of Concurrent Computation, Evians-les-Bains, Lecture Notes in Computer Science 70, Springer Verlag, 249-265.
4. Hamshere, B.C., 1983, 'A computer based environment for the design and analysis of concurrent systems : SIMULA implementation of the COSY notation', Proceedings of the 11th Ann. Conf. of the Association of SIMULA Users, Paris.

5. Janicki, R., Lauer, P.E., Devillers, R., 1983, 'Maximally concurrent evolution of non-sequential systems', Proceedings of the 4th European Workshop on Applications and Theory of Petri Nets, Toulouse, 188-202.
6. Shields, M.W., 1981, 'On the non-sequential behaviour of a class of systems satisfying a generalised free-choice property', Technical Report CRS 92-81, Computer Science Department, University of Edinburgh.
7. Lauer, P.E., Shields, M.W., Best, E., 1979, 'Formal Theory of the Basic COSY Notation', Technical Report 143, Computing Laboratory, University of Newcastle upon Tyne.
8. Best, E., 1982, 'Adequacy Properties of Path Programs', Theoretical Computer Science 18, North-Holland Publishing Co., 149-171.
9. Janicki, R., 1982, 'Partial orders and vectors of firing sequences', Report ASM/99, Computing Laboratory, University of Newcastle upon Tyne.
10. Shields, M.W., 1982, 'Non sequential behaviour 1', Technical Report CRS 120-82, Computer Science Department, University of Edinburgh.
11. Hillen, D., 1983, 'Adequacy-preserving substitution and reduction rules', Report ASM/108, Computing Laboratory, University of Newcastle upon Tyne.
12. Janicki, R., 1982, 'Transforming sequential systems into concurrent systems', Report ASM/93, Computing Laboratory, University of Newcastle upon Tyne.
13. Lauer, P.E., 1983, 'User's introduction to BCS : a computer based environment for specifying, analyzing and verifying concurrent systems', Report ASM/107, Computing Laboratory, University of Newcastle upon Tyne.
14. Lauer, P.E., Torrigiani, P.R., Shields, M.W., 1979, 'COSY : A system specification language based on paths and processes', Acta Informatica 12, Springer Verlag, 109-158.
15. Cotronis, J.Y., 1983, 'Programming and Verification of Asynchronous Systems', Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne.

16. Torrigiani, P.R., Lauer, P.E., 1977, 'Towards a macro notation for path expressions : examples of resource managing', AICA 1977, Annual Conference, 3rd Volume : Software Methodologies, 349-371.
17. Lauer, P.E., 1982, 'Realization of a high-level formalism in a computer based environment for designing distributed systems', Report ASM/104, Computing Laboratory, University of Newcastle upon Tyne.
18. Lauer P.E., Shields, M.W., 1980, 'COSY : An Environment for Development and Analysis of Concurrent and Distributed Systems', Proc. of Symposium on Software Engineering Environments, Lahnstein, June 1980, (Ed. H Hunke), North-Holland Publishing Co, 119-156.
19. Lauer, P.E., Shields, M.W., 1981, 'Interpreted COSY programs : Programming and Verification', Proceedings 2nd International Conference on Distributed Computing Systems, Paris, IEEE Computer Society Press, (Ed. E Gelenbe), 137-148.
20. Lauer, P.E., Shields, M.W., Cotronis, J.Y., 1981, 'Formal behavioural specification of concurrent systems without globality assumptions', Int. Colloq. on Formalization of Programming Concepts, Lecture Notes in Computer Science 107, Springer Verlag, 115-150.
21. Cotronis, J.Y., Lauer, P.E., 1983, 'Two way channel with disconnect', Proceedings of the SERC and STL workshop on Analysis of Concurrent Systems, Lecture Notes in Computer Science (to appear), Springer Verlag.
22. Shields, M.W., 1979, 'COSY train journeys', Report ASM/67, Computing Laboratory University of Newcastle upon Tyne.
23. Janicki, R., Lauer, P.E., 1982, 'Toward a solution of the Merlin-Randall problem of train journeys', Report ASM/95, Computing Laboratory, University of Newcastle upon Tyne.
24. Devillers, R., 1982, 'The train set strikes again', Report ASM/105, Computing Laboratory, University of Newcatsle upon Tyne.
25. Koutny, M., 1984, 'On the Merlin-Randall problem of train journeys', 6th International Symposium on Programming, Lecture Notes in Computer Science (to appear), Springer Verlag.

26. Li, W.L., Lauer, P.E., 1984, 'A VLSI implementation for COSY', Report ASM/121, Computing Laboratory, University of Newcastle upon Tyne.
27. Milner, R., 1980, 'A calculus of communicating systems', Lecture Notes in Computer Science 92, Springer Verlag.
28. Li, W.L., Lauer, P.E., 1984, 'Using the structural operational approach to express true concurrency', Report ASM/119, Computing Laboratory, University of Newcastle upon Tyne.
29. Plotkin, G., 1981, 'A structural approach to operational semantics', Lecture Notes, Aarhus University, Denmark.
30. Li, W.L., 1983, 'An operational approach to semantics and translation for concurrent programming languages', Ph.D. Thesis, University of Edinburgh.
31. Lauer, P.E., 1981, 'Synchronisation of concurrent processes without globality assumptions', New Advances in Distributed Computer Systems, NATO Advanced Study Institute Series C80, Reidel Publishing Co., 341-365.
32. Hoare, C.A.R., 1980, 'Synchronisation of parallel processes', Advanced Techniques for Microprocessor Systems, (Ed. Hanna, F.K.), Peter Peregrinus Ltd, 108-111.
33. Shields, M.W., Lauer, P.E., 1979, 'A formal semantics for concurrent systems', Proc. 6th Int. Colloq. for Automata, Languages and Programming, Graz, July 1979, Lecture Notes in Computer Science 71, Springer Verlag, 571-584.
34. Agerwala, T., 1977, 'Some extended semaphore primitives', Acta Informatica 8, Springer Verlag, 201-220.
35. Cotronis, J.Y., Lauer, P.E., 1979, 'On definitions of various notions of monitors in the COSY notation', Report ASM/58, Computing Laboratory, University of Newcastle upon Tyne.

Chapter 8

Ease of use through proper specification

Roger Gimson and Carroll Morgan

8.1 INTRODUCTION

In this chapter we describe a project whose aim is to construct a distributed operating system which would "make the programming and application of multiple microprocessor networks as simple and natural as the programming of single microprocessor workstations is today". In order to achieve this, we decided very early that it would be necessary to use in the construction the most up-to-date techniques for software specification, design, and development that were available to us. Our hope was that by doing this

we could use specifications to explore designs motivated purely by ease-of-use rather than by ease-of-implementation (since specification allows abstraction from implementation constraints),

we would have a precise notation in which such designs could be reliably communicated to others, and which would assist the discovery and discussion of the designs' implications,

it would be possible to present the specifications directly in the user manuals of our operating system, thus increasing their precision while decreasing their size, and

we could use the mathematical techniques of program refinement to produce implementations which were highly likely to satisfy their specifications (and hence would also be accurately described by their user manuals).

It seemed especially important that those benefits should be realised in the construction of a distributed operating system - because distributed operating systems offer the rare opportunity for the user to control the system, rather than vice versa. The high bandwidth of current local area networks allows efficient modularity; for example, a structure consisting of largely autonomous services and clients is entirely feasible. In such a system, the choice between (rival) services, and the manner in which they are used, would be entirely up to the clients. This is the basis of the open systems approach: provided services are well-specified, clients are free to make use of them in whatever manner is consistent with their specification.

8.2 A FIRST EXAMPLE

One of the most visible parts of any operating system is its file system. Even today, the design of these range in quality from excellent to horrific - in our opinion. Others may think instead that they range from horrific to excellent: that is, the features one user cannot

do without, another may abhor. It is through such features that an operating system controls (even the thoughts of) its clients, and this is exactly what we hope to avoid.

A file service in a distributed operating system is there to be shared by as many clients as possible. To achieve this, it must be unopinionated: it must have so few features that there is nothing anyone could object to. It is only in the context of specification that we can propose such a radical design; any less abstract context introduces efficiency constraints. Some of these, of course, will have to be met eventually, but perhaps not all of the ones that might conventionally be presumed necessary. We must not introduce such constraints simply because we could not express ourselves without them: first we state what we would like - then we compromise.

As an example, we propose the simplest file system design we could imagine. We describe it as a partial function files from the set NAME of file names to the set FILE of all possible files; we say nothing about the structure of the sets NAME and FILE themselves:

$$\text{files: NAME} \rightarrow \text{FILE}$$

The mathematical notation above introduces the variable files, and gives its type as NAME \rightarrow FILE. The English text states that this variable is to describe the file system. Our style of mathematical specification is an example of the Z specification technique, and we will continue to use it below. It is not possible for us to fully explain Z itself in this paper, but we hope its flavour will be evident; the bulk of the meaning will be conveyed by the English. Sufrin (9) and Morgan (4,5) together give an introduction to Z. The final section of this chapter provides a glossary of mathematical symbols.

We propose two operations only on the file system: StoreFile stores a (whole) file, and RetrieveFile (destructively) retrieves it.

StoreFile

Let files be the state of the file system before the operation, and let files' be the state afterwards. Let file? be the file to be stored, and let name! be some name, chosen by the file system, which will refer to the newly stored file (we conventionally use names ending in ? for inputs, and in ! for outputs). That is, given

$$\begin{aligned} \text{files, files': NAME} &\rightarrow \text{FILE} \\ \text{file?} &: \text{FILE} \\ \text{name!} &: \text{NAME} \end{aligned}$$

the effect of StoreFile is to choose a new name, which is not currently in use

$$\text{name!} \notin \text{dom files}$$

and to update the partial function so that after the operation, it maps the new name to the newly-stored file

$$\text{files}' = \text{files} \circ \{\text{name!} \mapsto \text{file?}\}$$

(We notice as an immediate advantage of our abstraction that we have given the implementer the freedom to store identical but differently named files in single or multiple copies, as he chooses.)

RetrieveFile

Let files be the state of the file system before the operation, and let files' be the state afterwards. Let name? be the name of the file to be retrieved, and let file! be the

file itself. That is, given

```
files, files': NAME → FILE
name?      : NAME
file!      : FILE
```

the effect of RetrieveFile is to return the named file to the client

```
file! = files (name?)
```

and to remove the name (and hence the file) from the partial function which represents the file system

```
files' = files / {name?}
```

The description above is of course infeasible with today's technology - which is a pity. It would be too inefficient to have to retrieve a whole file just to read one small piece of it. But how wonderful it would be if a file system could be so simple! We must take as our consolation that at least we were able to describe it.

8.3 THE FIRST COMPROMISES

The best we can do with our simple file system is to use it as the basis for a development of a more practical design - and the description above provides a context into which the necessary compromises can be introduced. Here are some of them (in no particular order):

Compromise

It must be possible to read the file without deleting it.

Clients must be prevented from destroying the files of others (remember, a file can't be updated).

Files must be given a limited lifetime, and clients must be charged for their storage.

Reason

The communication medium is not entirely reliable - a breakdown during retrieval could destroy the file without returning its contents.

Mistakes are inevitable - even honest clients could accidentally destroy another's file if not prevented.

Any implementation of the file system, however capacious, will still be finite.

We introduce these compromises in a revised design (again using the notation of Z largely without explanation). First, we name three new sets

CLIENT - the set of client identifications,

TIME - the set of instants (e.g. seconds from 1/1/80 - but we need not be specific here),


```

StoreFile
  ASS
  contents?: DATA
  expires? : TIME
  name!   : NAME
  cost!   : COST

  BFILE'. owner'   = who
         created'  = when
         expires'   = expires?
         contents'  = contents?

         name! ≠ dom files
         files' = files • [name! ↦ FILE']
         cost! = Tariff (FILE')
    
```

A new file FILE' is constructed which is owned by the client storing it, records its creation time as the time it was stored, which will expire at the time the client specified (then becoming inaccessible), and whose contents the client supplies.

A new name name! is chosen, not currently in use, and the file is stored under that name. The charge made is some function Tariff of the file (hence of its owner, creation and expiry times, and contents). Here is a possible definition of Tariff (which depends in turn on some function Size):

$$\text{Tariff} = \lambda \text{FILE}. (\text{expires} - \text{created}) * \text{Size}(\text{contents})$$

ReadFile

The ReadFile operation returns the expiry time and the contents of the file stored under a given name. Its parameters are the name of the file to be returned (name?), when it will expire (expires!), and its contents (contents!):

```

ReadFile
  ASS
  name?   : NAME
  expires!: TIME
  contents!: DATA

  SS' = SS

  BFILE. FILE      = files (name?)
         expires   > when
         expires!  = expires
         contents! = contents
    
```

ReadFile does not change the state of the service. The map files is applied to the name, to determine the file's value FILE, which must not have expired. Its expiry time and contents are returned.

DeleteFile

The DeleteFile operation removes a file from the service. A rebate is offered as an incentive to deletion before expiry. name? is the name of the file to be deleted, and cost! is the (possibly negative) charge made for doing so (we assume negation "-" is defined on COST):

DeleteFile	
ASS	
name?:	NAME
cost!:	COST
<hr/>	
FILE. FILE	= files (name?)
expires	> when
owner	= who
files'	= files \ {name?}
cost!	= - Rebate (FILE, when)

The map files is applied to the name, to determine the file's value FILE, which must not have expired. It must be owned by the deleting client. The file's name name? (and hence the file itself) are removed from the partial function which represents the stored files, and the cost is determined by a function Rebate of the file and its deletion time. Here is a possible definition of Rebate:

$$\text{Rebate} = \lambda \text{FILE}; \text{when}: \text{TIME}. (\text{expires} - \text{when}) * \text{Size}(\text{contents})$$

Naturally, there are other compromises which could be made, in addition to or instead of those above. In the next section, however, we discuss a compromise which we suggest should not be made.

8.4 A COMPROMISE AVOIDED

One glaring inefficiency remains in our proposal: that we must transfer whole files at once. Many clients will not have time or the resources (e.g. local memory) to do this. But here we will not compromise by modifying our file storage service to cater for this inefficiency - rather we insist that the business of the file storage service will be storage exclusively. Partial examination and updating will be the business of a file updating service.

To propose a service which treats the contents of files as having structure, we must propose a structure. The proposal we make is the very simple view that the contents of files is a sequence of pieces. (Sequences are functions from the natural numbers \mathbb{N} to their base type, and begin at index 1.) We do not say what a piece is, however.

DATA
seq PIECE

The file updating service in fact has no state; all its work is done in the calculation of its outputs from its inputs. Its two operations are ReadData and UpdateData.

ReadData

ReadData takes the contents of a file contents?, a starting position start?, and a number of pieces to be read number?, and returns the data pieces! at that position within contents?. (#pieces! is the length of the sequence pieces!, and 1..#pieces! is the set $\{i: \mathbb{N} \mid 1 \leq i \leq \#pieces!\}$.)

```

ReadData
  contents?: DATA
  start?,
  number? : IN
  pieces! : DATA
-----
  #pieces! = min (number?, (#contents? - start?))

  Vi: 1..#pieces!. pieces!(i) = contents?(i + start?)

```

The length of the data returned is equal to the number of pieces requested, if possible; otherwise, it is as large as the length of contents? will allow. The i -th piece of pieces! returned is equal to the $(i+start?)$ -th piece of contents?.

UpdateData

UpdateData takes the contents of a file contents?, a position start?, and some data pieces?, and returns an updated contents contents!.

```

UpdateData
  contents?,
  contents! : DATA
  start?   : IN
  pieces?   : DATA
-----
  #contents! = max (#contents?, (start? + #pieces?))
  start?     ≤ #contents?

  Vi: 1..#contents!.
    (i - start?) ∈ 1..#pieces? ⇒ contents!(i) = pieces?(i - start?)
    (i - start?) ∉ 1..#pieces? ⇒ contents!(i) = contents?(i)

```

The length of the new contents is equal to its original length, unless an extension was necessary to accommodate the new data; however, the new data must begin within the original contents or immediately at its end. The i -th piece of contents! is equal to the $(i-start?)$ -th piece of pieces?, if this is defined; otherwise, it is equal to the i -th piece of contents?.

Our proposal is of course only one of the many possible (for example, see the definition of these operations in Morgan and Sufrin (6)). We could, of course, propose several updating services, each providing its own set of facilities. Moreover, the original operations which transferred whole files would still be available to those clients able to use them. (See figure 81).

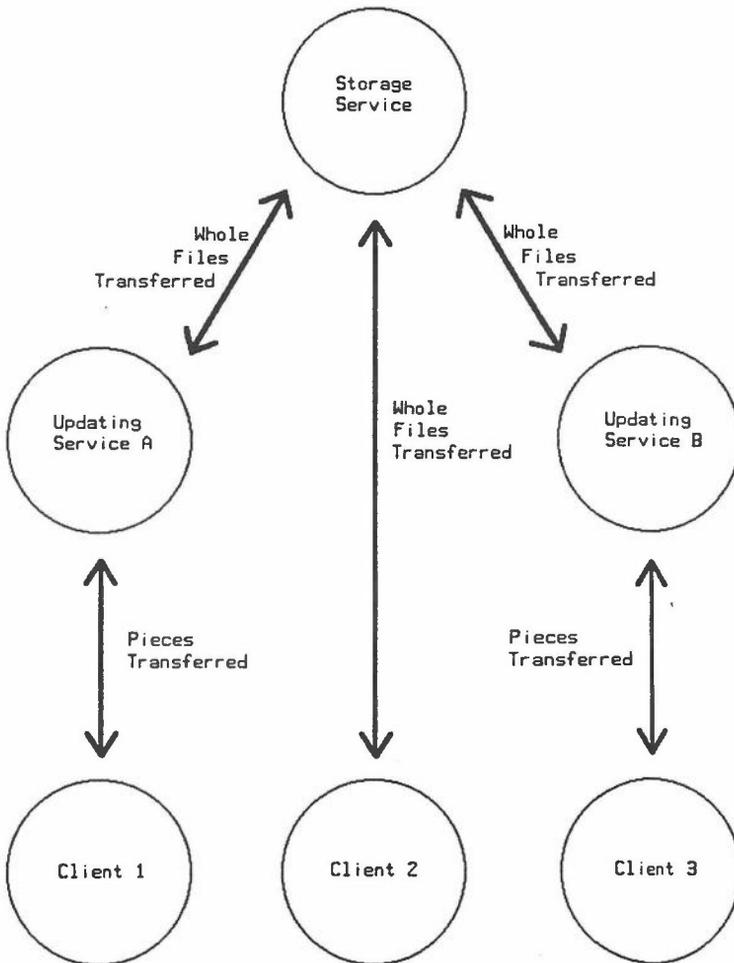


Figure 8.1

8.5 MODULARITY AND COMPOSITION OF SERVICES

The structure we have presented above separates the issues of how files should be stored from how they should be manipulated. As a result, we have offered the user an unusual freedom of choice - he can read just one piece of a file, or he can treat a file as a single object (with the corresponding conceptual simplification; Stoy and Strachey (8) for example allow this in their operating system OS6).

Still, it's likely that a further compromise will be necessary: for large files, the time taken to transfer the file between the two services (storage and updating) may not be tolerable. We solve this not by changing our design, but by an engineering decision: for applications that require it, we will provide the two services together in one box, and the transfers will be internal to it. (See figure 8.2.) Its specification we construct by combining the material already available.

StoreFile, ReadFile, and DeleteFile will be available as before. We introduce two new operations, however - ReadStoredFile, and UpdateStoredFile - whose specifications will be formed by composing the specifications given above. (The schema composition operator \wr , used for this, is defined in (4). Here, we will explain it informally.)

ReadStoredFile

Reading a stored file is performed by first reading the whole file with ReadFile, and then reading the required portion of its contents using ReadData. In Z we write this

ReadStoredFile \wr ReadFile ; ReadData

If we were to expand this definition of ReadStoredFile, the result would be as below:

```

ReadStoredFile
-----
    ASS
    name? : NAME
    start?,
    number? : IN
    expires!: TIME
    pieces! : DATA

    SS' = SS

    3FILE. FILE      = files (name?)
        expires > when
        expires! = expires

        #pieces! = min (number?, (#contents - start?))
        Vi: 1..#pieces!. pieces!(i) = contents(i + start?)
    
```

ReadStoredFile takes a file name name?, a starting position start?, and a number of pieces number?, and returns the expiry time of the file expires!, and the data pieces! found at the position specified. (expires! is returned by ReadStoredFile because ReadFile returns it; we could have dropped this extra output, but choose not to introduce the Z notation for doing so.)

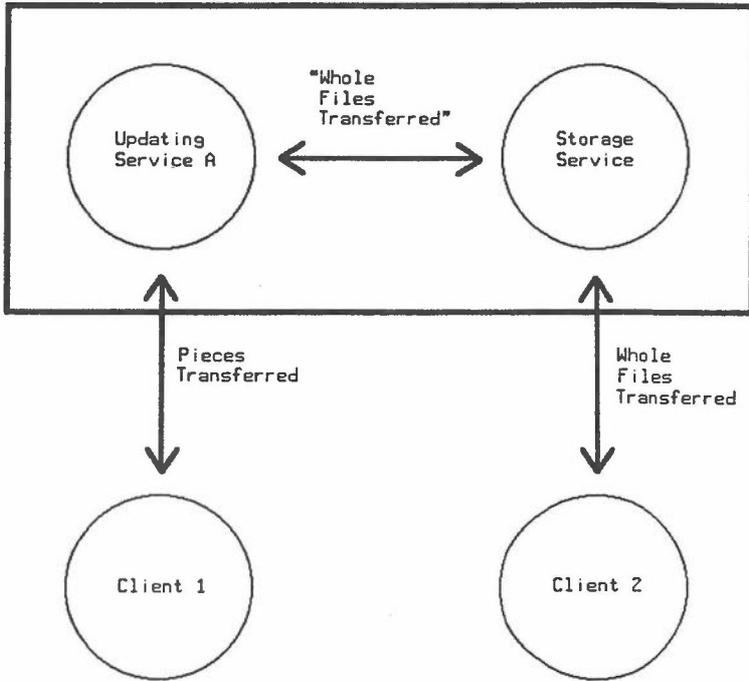


Figure 8.2 Updating and storage service

UpdateStoredFile

The complementary operation `UpdateStoredFile` is a more difficult composition, since we must accumulate the costs of the component operations, and we must ensure the updated file is (re-)stored under its original name. For the sake of honesty, we will give the definition, but we will not expand it:

```
UpdateStoredFile  a
    ReadFile ;
    DeleteFile [dcost!/cost!];
    UpdateData ;
    StoreFile [name?/name!, scost!/cost!];
    (dcost?, scost?, cost!: COST | cost! = dcost? + scost?)
```

`UpdateStoredFile` first reads the whole file, then deletes it, then updates it, and then stores its new value under its original name. Finally, it presents as its overall cost the sum of the two charges made by `DeleteFile` (which may well be negative) and `StoreFile`.

Machine assistance in the manipulation of expressions such as the above is the subject of a separate research project within the Programming Research Group (the Software Engineering Project).

What we have done is to compose two simple but infeasible operations to produce a more complicated but feasible one (rather like the use of complex numbers in electrical engineering, for example). Naturally, the implementor will not transfer whole files back and forth within his black box on every read and update operation - but nevertheless the updating and storage service provided by the box must behave as we have specified. Our decomposition was chosen for economy of concept; the implementor's must be chosen for economy of time and equipment, and the whole range of engineering techniques are available to him to do so (caches, update-in-place, etc.).

8.6 EXPERIENCE SO FAR

While we have followed the general principles above, we have in fact adapted to constraints in different ways. Our storage service, which we have implemented in prototype, stores blocks of a fixed size (rather like the service described in Biekert (1)). This distinguishes our "universal" storage service from, say, the one implemented at Cambridge (described in Needham (7)). Organisation of blocks into files, the keeping of directories, etc. is done by software in the clients' own machines (for example, using a "File Package" (Gimson (2))). This allows clients freedom in the choice of what file structure they build, but of course makes the sharing of files more difficult. If one package should become popular, it could be placed in a machine of its own, and so become a service.

It has not been possible to cover many aspects of our work in this presentation. For example, the specification of the errors that may occur in use is an essential part of the full specification of a service, and is included in service user manuals (for example see Morgan (3)).

We can't claim to have made "the programming and application of multiple microprocessor networks as simple and natural as the programming of single microprocessor workstations" - not yet. So far, the pressure of simplicity in our mathematical descriptions has kept our designs correspondingly simple. At present, they are perhaps too much so; but by using "the most up-to-date techniques for software specification..." we have built basic services which genuinely are simple. And that is where one must begin.

8.7 FUTURE PLANS

The styles of specification, and of presentation of user manuals, has to some extent been developed in parallel with the software to which they have been applied. These styles are now more stable, and we intend to specify, design, and implement further services in the same way.

Our goal is to produce a suite of designs from which implementations can be built on a variety of machines. Each design will be documented, in our mathematical style, both for the user and for the implementor. Thus we can say that our primary goal is to construct a distributed system on paper.

For our paper construction to have any value, the designs proposed in it must be widely applicable, and genuinely useful. We expect our machine-independent techniques of description to take care of the first requirement. To ensure that the second is met, we must construct prototype implementations of each of the designs, and we must gain experience of their use.

8.8 GLOSSARY OF SYMBOLS

$\hat{=}$	"is syntactically equivalent to"
\mathbb{N}	The set of natural numbers (non-negative integers)
$\{sig \mid pred\}$	The set of sig such that $pred$
$m..n$	The set of natural numbers between m and n inclusive
	$m..n \hat{=} \{k: \mathbb{N} \mid m \leq k \leq n\}$
$A \rightarrow B$	The set of partial functions from A to B
$[a \mapsto b]$	The function $\{(a,b)\}$ which takes a to b
$f(x)$	The function f applied to x
dom	The domain of a relation (or function)
	for $f: A \rightarrow B$,
	$dom f \hat{=} \{a: A \mid (\exists b: B . b = f(a))\}$
\setminus	Domain co-restriction
	for $f: A \rightarrow B$; $\hat{a} \subseteq A$,
	$f \setminus \hat{a} \hat{=} \{(a,b): f \mid a \notin \hat{a}\}$
\circ	Functional overriding
	for $f, g: A \rightarrow B$,
	$f \circ g \hat{=} (f \setminus dom g) \cup g$
$seq A$	The set of sequences whose elements are drawn from A
	$seq A \hat{=} \{s: \mathbb{N} \rightarrow A \mid (\exists n: \mathbb{N} . dom s = 1..n)\}$

#s The length of sequence s
 dom s = 1..#s

[new/old] Schema variable renaming
; Schema forward composition

REFERENCES

1. Biekert, R., and Janssen, B., 1983, "The implementation of a file system for the open distributed operating system Amoeba", Informatica Rapport, Vrije Universiteit, Amsterdam.
2. Gimson, R. B., 1983, "A File Package - User Manual", Distributed Computing Project Working Paper, Programming Research Group, Oxford University
3. Morgan, C. C., 1983, "A Block Storage Service - User Manual", Distributed Computing Project Working Paper, Programming Research Group, Oxford University
4. ---, 1984, "Schemas in Z - a preliminary reference manual", Distributed Computing Project Working Paper, Programming Research Group, Oxford University
5. ---, 1984, "Schemas in Z - an example", Distributed Computing Project Working Paper, Programming Research Group, Oxford University
6. ---, and Sufrin, B., 1984, "Specification of the Unix File System", IEEE Trans. Soft. Eng., March 1984
7. Needham, R., and Herbert, A., 1982, "The Cambridge file service", in "The Cambridge Distributed Computing System", Addison-Wesley, 41-63
8. Stoy, J. E., and Strachey, C., 1972, "OS6 - An operating system for a small computer", Comp. J. 15, 2, 195-203
9. Sufrin, B., 1983, "Mathematics for system specification", Lecture Notes 1983-1984, Programming Research Group, Oxford University

Probabilistic modelling of distributed computing systems

I. Mitrani

9.1. GENERAL BACKGROUND

There are three main methods for evaluating the performance of a complex computer system. These are (a) observing the system in operation and taking measurements, (b) writing and running a simulation program that mimics the behaviour of the system and (c) constructing and solving a mathematical model that captures the essential features of the system. The last of these methods is in many ways the most satisfying, both intellectually and from the practical point of view. Its chief advantage is that it provides deep insights into underlying trends and functional relationships between system parameters and performance characteristics. Also, mathematical modelling requires little or no computing resources; hence, it is usually very cheap. On the other hand, a mathematical model is, by its nature, only an approximation of reality; moreover, it is often necessary to make further simplifying assumptions in order to ensure numerical or analytical tractability. The application of the modelling approach to the performance evaluation of different types of distributed computing systems, and the accompanying trade-offs between accuracy and cost, is the subject of this article.

Systems whose behaviour is influenced by random phenomena - and computer systems certainly fall in that category due to the unpredictability of the demands placed upon them - are modelled by means of stochastic processes. A stochastic process is a random function of time whose value at any given moment is a possible system state. In the cases that interest us, the system states are usually represented by vectors of integers, and so the associated stochastic processes can be thought of as random walks on multidimensional lattices. For example, in a system consisting of two computers connected by a communication line, allowing jobs to be submitted and executed at either site, the system state at time t might be described by a pair of integers, $[n_1(t), n_2(t)]$, where $n_i(t)$ is the number of jobs waiting and/or being executed at site i ($i=1,2$). The system could then be modelled by a two-dimensional stochastic process $\{[n_1(t), n_2(t)] ; t \geq 0\}$. The state lattice and the possible one-step transitions for such a process are illustrated in Fig. 9.1.

To specify a model completely, it is of course necessary to make assumptions governing the probabilistic behaviour of the associated stochastic process. From the point of view of the analysis, it is highly desirable (in most cases imperative) that the process possesses the so called 'Markov' or 'memoryless' property: The states entered after a given moment, t , may depend on the state at t , but not t itself or on anything that happened before t . The Markov property is closely

related to the exponential distribution. The time that a Markov process spends in any given state, on any visit to that state, is distributed exponentially (with parameter which may depend on the state). In order to ensure that this property holds, various random variables that affect the behaviour of the process - such as intervals between job arrivals, job execution times, etc. - are assumed to be distributed exponentially.

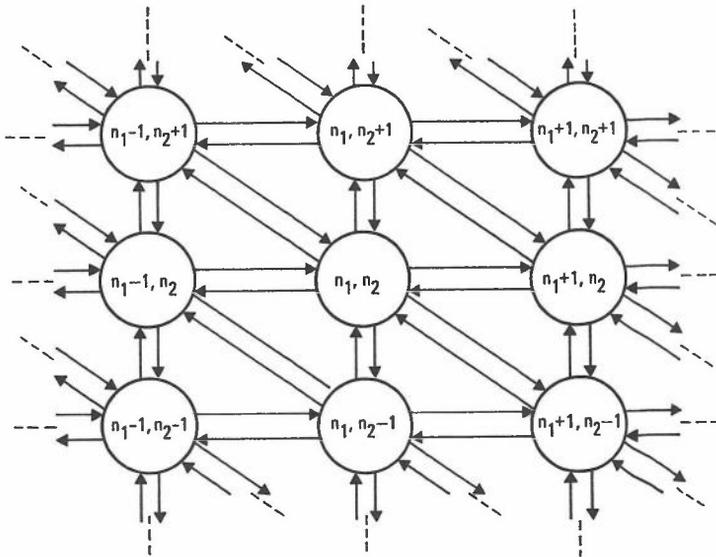


Fig. 9.1. State transition lattice

We are interested mainly in the long-run, or equilibrium, or steady-state behaviour of a system. For a Markov process whose state vector, $\underline{S}(t)$, takes values, \underline{s} , in some denumerable set \mathcal{G} , the steady-state distribution is defined by the limits

$$p(\underline{s}) = \lim_{t \rightarrow \infty} P [S(t) = \underline{s} \mid S(0) = \underline{s}_0], \quad \underline{s}, \underline{s}_0 \in \mathcal{G} \quad (9.1)$$

provided that the latter exist for all states \underline{s} in \mathcal{G} , are independent of the initial state \underline{s}_0 and sum up to 1. When it exists, the steady-state distribution and hence other performance measures, can in principle be determined by solving a system of linear equations (e.g. see Gelenbe and Mitrani (6))

$$p(\underline{s}) \left[\sum_{\underline{s}' \in \mathcal{G}} r(\underline{s}, \underline{s}') \right] = \sum_{\underline{s}' \in \mathcal{G}} [p(\underline{s}') r(\underline{s}', \underline{s})], \quad \underline{s} \in \mathcal{G} \quad (9.2)$$

The quantities $r(\underline{s}, \underline{s}')$, $\underline{s}, \underline{s}' \in \mathcal{G}$, referred to as the 'instantaneous transition rates', are known; they are simple functions of basic model parameters such as job arrival rates, average execution times, numbers of processors and terminals, etc. It is intuitively appealing

to interpret $r(\underline{s}, \underline{s}')$ as the average number of transitions that the process makes from state \underline{s} to state \underline{s}' , per unit time that it spends in state \underline{s} . With that interpretation, equations (9.2) become almost obvious: they simply state that the average number of transitions out of state \underline{s} per unit time is equal to the average number of transitions into state \underline{s} per unit time, for all \underline{s} . For that reason, (9.2) are known as the 'steady-state balance equations'.

Thus the general approach to the analysis of models which are formulated in terms of a Markov process involves writing a set of balance equations and then finding the unique solution $p(\underline{s})$, \underline{s} whose elements sum up to 1. However, while the first of these steps is usually quite easy, solving the equations can be very difficult indeed. In particular, if the state space (and hence the number of equations) is infinite, close-form solutions are available only in a few special cases; some of these will be described later. Even when the state space is finite, but large, deriving an exact solution is often too expensive to be practical. It is important, therefore, to consider ways of obtaining good approximate solutions.

One approximation technique that has proved very useful in the context of distributed computer systems consists of expressing certain essential quantities in terms of themselves. This is done by making simplifying assumptions about the dependencies between system components isolating subsystems which are easily solvable and feeding the solutions for those subsystems back into the model. The problem is then reduced to a 'fixed-point' equation of the type $x = f(x)$, where x is either a scalar or a vector and is related either directly or indirectly to the performance of interest (de Souza e Silva et al (3), Mitrani, (14)). Such an approximation can, of course, be applied even when the underlying stochastic process is not Markov.

The following section is devoted to models that can be solved exactly. The applications include multi-processors systems (with and without breakdowns) and networks where the interactions between jobs and processors satisfy a particular set of assumptions.

Section 3 deals with cases where those assumptions are violated, and where only approximate solutions are available. Distributed data bases and local area communication networks fall in that category. Before proceeding, however, it is worth mentioning here a fundamental result which will be very useful later.

Consider an arbitrary system where jobs arrive, spend some time and then depart. The internal composition of the system is unimportant but it should be in the steady-state. Let λ be the average number of jobs entering (and leaving) the system per unit time, W be the average period that jobs spend in it and L be the average number of jobs present in it at a given moment. Then these three quantities satisfy the relation

$$L = \lambda W \quad (9.3)$$

regardless of the nature of any random variables that are involved, and whether they are independent or not. This is known as 'Little result', (13). It implies that, for a given throughput, the congestion and the response time are essentially equivalent as measures of performance.

9.2. MODELS THAT CAN BE SOLVED EXACTLY

All systems examined in this section are modelled by Markov processes. We shall classify them according to their dimensionality, i.e. the number of elements in the vector describing the system state.

9.2.1 Multiprocessor Systems with a Single Job Type

Suppose that all jobs submitted for execution at a computer system are of the same type, i.e. they are statistically identical. The computing resources consist of N identical processors, each of which executes jobs, one at a time, to completion. When there are more than N jobs in the system, N of them are being executed, while the rest wait in a common queue. To ensure that the memoryless property holds, assume that the arrival instants form a Poisson stream (with rate λ per unit time) and that job execution times are exponentially distributed (with mean $1/\mu$). Then the system state at time t is completely described by a single integer, $n(t)$, specifying the number of jobs that are waiting and/or being served at that time. Moreover, $\{n(t), t \geq 0\}$ is a Markov process which attains steady-state if the available processors can cope with the work submitted, i.e. if $(\lambda/\mu) < N$.

This model is known in the queueing theory literature as the $M/M/N$ system (Markov arrivals, Markov service, N servers). If the process is in state n , the only possible one-step transitions are to state $n+1$ (if the next event is the arrival of a new job), or to state $n-1$ (if the next event is a departure and $n > 0$). Markov processes with a transition structure of this type are called 'one-dimensional Birth-and-Death processes'. In our case, the average number of transitions from state n to state $n+1$ per unit time that the process spends in state n , i.e. the rate of birth in state n , is equal to λ for all $n \geq 0$: Similarly, the rate of death in state n is equal to $\mu_n = \min(n\mu, N\mu)$, since the departure rate is proportional to the number of busy processors. The balance equations are easy to solve (see (6)), yielding the steady-state distribution of the process and such performance measures as the utilisation of the processors, average queue size and (using Little's result), average response time. This analysis shows, for example, that if the number of processors increases and their speed decreases, so that the total processing capacity $N\mu$ remains constant, the system performance deteriorates. Thus, other things being equal, a single processor is best.

Now consider the more interesting case where the processors are not completely reliable. Each processor goes through alternating periods of being operative and broken down, independently of the others. The operative and inoperative periods are distributed exponentially, with means $1/\xi$ and $1/\eta$ respectively. Thus each processor is operational for a proportion $\eta/(\xi + \eta)$ of the time and hence the effective number of processors available is $N\eta/(\xi + \eta)$.

The system state at time t is described by a pair of integers $[n(t), m(t)]$, where $n(t)$ is the number of jobs present and $m(t)$ is the number of operative processors. There is still essentially a one-dimensional process because $m(t)$ takes only a finite number of values, $0, 1, \dots, N$. The condition for existence of steady-state is again that the submitted load is less than the available processing power: $(\lambda/\mu) < N\eta/(\xi + \eta)$. However, the balance equations are more complicated than in the previous case and their solution is considerably more difficult (Mitrani and Avi-Itzhak (15), Neuts and Lucantoni (17)). Without going into details, the steady-state distribution of the system state (n, m) is obtained indirectly by determining N one-variable generating functions. The derivatives of those generating functions at point 1 yield the average number of jobs present and hence the average response time.

It turns out that, in the presence of this type of unreliability, the optimal number of processors is no longer always 1; in general, it

is greater than 1. Moreover, the closer the system is to saturation, i.e. the closer the submitted load (λ/μ) to the available processing power $[N\eta/(\xi + \eta)]$, the greater is the optimal number of processors.

The above model can be generalised in several directions. Processors might be more likely to breakdown when they are busy than when they are idle. There could be a finite number, R, of repairmen ($R \leq N$), so that a broken down processor may have to wait before its repair can start. The distributions of operative and repair periods may be other than exponential (this last generalisation is much more expensive than the others).

A different problem in the area of multiprocessor systems concerns the extent to which internal parallelism can be exploited in the execution of a job. One can construct a model of a job as a collection of tasks, some of which can be executed in parallel on different processors. The question then would be: given the structure of that collection, how long does it take a certain number of processors to execute the job? Some results in that connection were obtained in Fayolle et al (5); we shall not dwell on them here because both the model and the solution techniques fall outside the present framework.

9.2.2 Two Job Types

A fundamentally different problem arises if the jobs arriving into our multiprocessor system may belong to two different types, with (perhaps) different arrival rates and average execution times. The computer manager may wish to give unequal treatment to the two job types and provide one of them with a better service than the other. One way of doing this is to designate K of the N processors as 'type 1' and the other N-K as 'type 2'. Type i jobs can then be given pre-emptive priority on type i processors ($i=1,2$). Thus the processing capacity available to type 1 jobs is K when there at least N-K type 2 jobs in the system, but can rise (up to N) when that number drops below N-K. The scheduling strategies defined in this manner vary from 'absolute priority for type 1 jobs' (when $K=N$), to 'absolute priority for type 2 jobs' (when $K=0$).

Under exponential assumptions, this system is modelled by a two-dimensional Birth-and-Death process $\{[n_1(t), n_2(t)] , t \geq 0\}$, where $n_i(t)$ is the number of type i jobs present in the system at time t. The condition for existence of steady-state is $(\lambda_1/\mu_1) + (\lambda_2/\mu_2) < N$, where λ_i and $1/\mu_i$ are the arrival rate and the average execution time, respectively, for type i jobs ($i=1,2$).

The state transition structure of this process is easy to describe: from state (n_1, n_2) , the process moves to

state (n_1+1, n_2) with rate λ_1

state (n_1, n_2+1) with rate λ_2

state (n_1-1, n_2) with rate $\mu_1 \min [n_1, K + \min(0, \min(0, N-K-n_2))]$,
for $n_1 > 0$

state (n_1, n_2-1) with rate $\mu_2 \min [n_2, N-K + \min(0, K-n_1)]$,
for $n_2 > 0$

The solution of the set of balance equations reduces to that of a functional equation in two variables. The 'mixed priority' scheduling strategies ($0 < K < N$) are more difficult to deal with than the 'absolute

priority' ones ($K=0$ or $K=N$); in the former case, the functional equation is further reduced to a boundary problem on a closed contour. In both cases, the solutions were obtained quite recently (Fayolle et al (4), Mitrani and King (16)).

In the context of systems with multiple job types it is reasonable to enquire to what extent an improved performance for one job type is paid for by a deterioration in performance for another type. If performance for type i is measured by the average number of jobs of that type in the system, L_i (or equivalently, by the average time that they have to wait), and if service is provided by a single processor, then the answer to that question is supplied by a result known as 'Kleinrock's conservation law' (Kleinrock (12)). The latter states that any decrease in the value of one or more of the L_i 's, by means of a change in the scheduling strategy, is compensated by an increase in other L_i 's, in such a way that the linear combination

$$C = \sum_i (L_i / \mu_i) \quad (9.4)$$

remains constant (the sum is taken over all job types). Whether that conservation law holds for multiprocessor systems is, at present, an open problem. Some numerical experimentation with the above two job types model seems to indicate that it holds, at least approximately. For a given number of processors, N , the scheduling strategy was varied by varying K from 0 to N and the linear combination (9.4) was evaluated each time. Some differences were observed, but they were so slight that they could have been explained by accumulations of round-off errors, rather than by failure of the conservation law.

9.2.3. Network Models

When a system comprises several interconnected service stations (which may be processors, I/O devices, etc.) offering different types of service to jobs that may move from one station to another, the appropriate model to use is likely to be a queueing network. To define such a model, one has to specify (a) the set of nodes and their characteristics, and (b) the behaviour of the jobs. As usual, there is a fine dividing line between the assumptions that lead to tractable models and those that do not.

One of the first important results in the area of queueing networks was obtained by J.R. Jackson (9) more than twenty years ago; the models to which that result applies are hence known as 'Jackson networks' (see also (6)). A Jackson network consists of N nodes numbered $1, 2 \dots N$, and a single job type. The execution times at node i are exponentially distributed with mean $1/\mu_i$ (that parameter may also be assumed dependent on the number of jobs at node i). Jobs arrive into the network in a Poisson stream at rate λ per unit time. A newly arrived job joins node i with probability q_{0i} ($i=1, 2, \dots, N$). After completing execution at node i , jobs go to node j with probability q_{ij} ($i, j=1, 2, \dots, N$); they leave the network with probability q_{i0} . The matrix of probabilities q_{ij} is called the 'routing matrix'.

The state of the network at time t is described by the vector $[n_1(t), n_2(t), \dots, n_N(t)]$, where $n_i(t)$ is the number of jobs at node i . It might seem, in view of the difficulty in analysing the two-dimensional processes of the previous sub-section, that this N -dimensional Markov process would prove to be at least as hard to solve. This, however, is not so: Jackson networks turn out to have a surprisingly

simple solution.

Let λ_i be the average number of jobs coming into node i per unit time (from outside and from other nodes). If the network is in steady-state, λ_i is also the average number of jobs leaving node i per unit time. These throughput rates must satisfy the following system of 'flow balance' equations.

$$\lambda_i = \lambda \alpha_{0i} + \sum_{j=1}^N \lambda_j \alpha_{ji}, i = 1, 2, \dots, N \quad (9.5)$$

When the system (9.5) has a unique solution, the network is said to be 'open'. Intuitively, this means that jobs do come in from the outside ($\lambda > 0$), and no job remains in the network forever. Jackson's result states that, in an open network in the steady-state, the numbers of jobs at the different nodes are independent Birth-and-Death processes. In other words, the probability that the network is in state (n_1, n_2, \dots, n_N) is equal to the product of the probabilities $p_i(n_i)$, of finding n_i jobs in a single node system with Poisson arrivals (rate λ_i) and exponential execution times (mean $1/\mu_i$) for $i=1, 2, \dots, N$:

$$p(n_1, n_2, \dots, n_N) = p_1(n_1)p_2(n_2)\dots p_N(n_N) \quad (9.6)$$

Thus, having determined the throughputs λ_i from (5), node i is treated as a one-dimensional Birth-and-Death process with parameters λ_i and μ_i , in isolation of the other nodes. The known solutions for these one-dimensional processes are then multiplied together, to obtain the joint distribution of the network state (9.6). Performance measures such as average numbers of jobs or average sojourn times are easily calculated. For a given node, these come from the one-dimensional solution $p_i(n_i)$. For a group of nodes, or the whole network, the developments are only slightly more complicated. Suppose, for example, that we are interested in the average times, Q_i , that a job spends in the network after leaving node i ($i=1, 2, \dots, N$); also, in the average total time spent in the network, Q_0 . These quantities satisfy a system of linear operations:

$$Q_i = \sum_{j=1}^N \alpha_{ij}(W_j + Q_j)$$

where W_j is the average sojourn time at node j .

Jackson's result is remarkable in several aspects. First, it is counterintuitive : the nodes obviously influence each other, so the queue sizes could be expected to be dependent. Second, individual nodes behave as though they were subjected to a Poisson input of jobs, whereas the total input stream into a node is not in general Poisson. Third, the independence of the queue sizes is in marked contrast with the fact that the sojourn times of a job at the nodes that it visits are, in general, dependent. To illustrate this last phenomenon, consider the simple three-node network in Fig. 9.2:

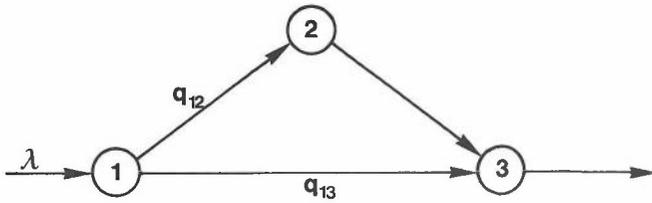


Fig. 9.2. A three-node network

If a job goes from node 1 directly to node 3, its sojourn times at the two nodes are independent of each other; if, however, that job goes via node 2, then its sojourn times at nodes 1 and 3 are dependent. What causes that dependence is the possibility of the job being overtaken by other jobs which were behind it at node 1. The distribution (as opposed to the average) of sojourn times along paths that allow overtaking is, except in some special cases, an open problem.

If no jobs arrive into a network from outside and no jobs leave it (i.e. $\lambda=0$ and $q_{i0}=0$ for all $i=1,2,\dots,N$), then that network is said to be 'closed'. Closed networks can be used to model distributed systems under heavy load, when there is a certain number of jobs circulating among the service stations at all times.

The flow balance equations (9.5) are still satisfied in a closed network, but since they are now homogenous, they no longer determine uniquely the node throughputs λ_i . Nevertheless, closed exponential networks have a product-form solution similar to (9.6):

$$p(n_1, n_2, \dots, n_N) = G p_1(n_1) p_2(n_2) \dots p_N(n_N) \quad (9.7)$$

where $p_i(n_i)$ is obtained by using any solution of (9.5) and G_i is chosen so that the right hand sides of (9.7), when summed over all network states, add up to 1. This result was established by Gordon and Newell (8) and bears their names.

The number of possible states for a closed network with N nodes and K jobs circulating amongst them, is equal to $\frac{[(N+K-1)!]}{[K!(N-1)!]}$. Since that number grows rather quickly with N and K , the computation of the 'normalising constant' G in (9.7) is a non-trivial task for networks of realistic size. Fortunately, a number of efficient numerical algorithms are available for that purpose. Having computed the constant, other performance measures are easily obtained.

The queueing network model can be generalised considerably without losing the product form solution (see (6)). An important generalisation is to assume that jobs may be of different types and may change type as they move from node to node. Thus the routing of jobs can proceed according to probabilities $q_{ir,js}$ that a job of type r leaving node i will go to node j as a job of type s . The matrix of these routing probabilities need not be irreducible; it is possible for jobs of different types to circulate among different groups of nodes. Also the network may be open with respect to some job types and closed with respect to others.

When jobs belong to different types, one may wish to use scheduling strategies that treat them differently, e.g. discriminate in favour of

one or more job types and against others. It turns out that if this is done, the product form of the solution is lost and the model becomes intractable. The scheduling strategies that can be allowed, such as First-In-First-Out, Last-In-First-Out or Processor-Sharing, treat all jobs equally (at FIFO nodes, it is even necessary that all job types have the same average execution times). There have been a number of studies involving network models with priority scheduling strategies, but they have always had to seek approximate solution methods.

A few words on the robustness of the network results are perhaps in order. As a general rule, the more specific the required performance measure, the more sensitive it is with regard to the assumptions of the model. Thus quantities like node utilisations and throughputs in open networks depend only on the average interarrival and execution times, and not on their distributions. As far as average numbers of jobs or sojourn times are concerned, the exponential assumptions often provide very good approximations. The variance and higher moments of those measures (at FIFO nodes) tend to be more affected by the shape of the distributions.

9.3 MODELS THAT REQUIRE APPROXIMATIONS

In principle, any model can be solved exactly, or at least to any degree of accuracy. For that, it suffices to make the state space finite, if it is not so already, and then apply a numerical algorithm. In practice, of course, such an approach is very likely to be prohibitively expensive, which is what is meant when the model is said to be 'intractable'. However, the fact that a model is intractable does not necessarily preclude the possibility of obtaining perfectly reasonable estimates of the desired performance measures with very little effort. The unfeasible exact solution can often be replaced by an easily computable approximate one.

The general idea in deriving approximate solutions is to reduce the complexity of the model by examining certain parts of it in isolation and, having solved the corresponding simpler models, use the results to evaluate the interaction between the parts. An approach that seems to work well for distributed systems proceeds as follows: By analysing an appropriately chosen (and easily solvable) sub-system, obtain certain of its characteristics - call them X - as a function of certain parameters - call them Y - approximately representing the effect of the rest of the system on the sub-system. This yields a relationship of the form $X = F(Y)$. Next, expressing the interaction parameters Y in terms of the sub-system characteristics produces a 'complementary' relationship $Y = G(X)$, which again may be approximate. Those relations lead to a fixed-point equation, $X = H(X)$, which can be solved for X .

As an illustration of this idea, consider a closed cyclic network (Fig. 9.3) with N nodes and K jobs circulating among them (there is no product form solution if queueing at each node is FIFO and execution times are not exponential). (see Fig. 9.3 Next Page)

Let W_i be the average sojourn time of a job at the i 'th node, L_i be the average number of jobs at the i 'th node, and λ be the throughput of the network (i.e., the average number of jobs passing through any node per unit time). Assuming that L_i is also the average number of jobs at node i that a job sees when it gets there, we can write an approximate expression

$$W_i = (L_i + 1)s_i, \quad i = 1, 2, \dots, N, \quad (9.8)$$

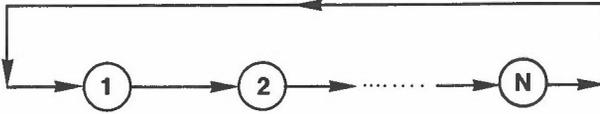


Fig. 9.3 A closed cyclic queueing network

where s_i is the average execution time at node i . To obtain a complementary expression for the L_i 's in terms of the W_i 's, note that the average time for a job to make a complete cycle is $(W_1 + W_2 + \dots + W_N)$, hence the rate at which a given job passes through a node is the reciprocal of that sum. Since there are K jobs altogether, the network throughput is given by

$$\lambda = K / \sum_{i=1}^N W_i$$

Little's result now provides an expression for L_i :

$$L_i = \lambda W_i = KW_i / \sum_{j=1}^N W_j, \quad i = 1, 2, \dots, N \quad (9)$$

Finally, (9.8) and (9.9) yield a set of fixed point equations for the W_i 's:

$$W_i = S_i \left[1 + KW_i / \sum_{j=1}^N W_j \right], \quad i = 1, 2, \dots, N \quad (10)$$

Arguments very similar to the above form the basis of a general method for the analysis of closed queueing networks. This is known as 'Mean Value Analysis', or MVA (Reiser and Lavenberg (18), Bard (1)). It is possible to improve significantly the approximation involved in writing (9.8) or even, under certain assumptions, to replace (9.8) with an exact relation (in the latter case, the resulting equations are recursive, rather than fixed-point).

The 'natural' way to solve equations of the form $X = H(X)$, regardless of whether X is a scalar or a vector, is by iteration: start with an initial value, X_0 , and at the n 'th step compute $X_n = H(X_{n-1})$, until the sequence X_n converges. Unfortunately, it is possible that the fixed-point equation has a unique solution, and yet the iteration sequence does not converge. When that happens, other solution methods have to be employed.

The models that are described in the following have this in common: they are intractable as far as exact solutions are concerned, but lend themselves quite easily to approximation by the fixed-point methods.

9.3.1 A Distributed Data Base Model

Consider a data base consisting of a large number, D , of items of information, and suppose that it can be accessed in parallel by N statistically identical users. At any moment in time, a user is either passive (in 'think' state), or is executing a transaction; these two states strictly alternate. During its execution, a transaction may re-

quire data items. These are acquired dynamically as requested and are held until either the execution is successfully completed, or the transaction is aborted. An item cannot be held simultaneously by more than one transaction. If, at the time when a transaction requires an item, the latter is not available, that transition goes into a wait state and tries again later. In order to recover from deadlocks, a limit on the number of such retries is imposed; when that limit is reached, the requesting transaction is aborted, it releases all items acquired so far and has to restart its execution from the beginning.

In order to describe the state of the system, one has to specify what each user is doing (passive, running, waiting for an item, etc.) and what is the location of each item. It is clear that, even under Markovian assumptions, the problem is intractable for any but a few small values of N and D . To obtain an approximate solution, consider a single user in isolation and assume that the influence of all other users on him is stationary. That influence is manifested by two parameters whose values are as yet unknown: F and F_1 . F is the probability that when a transaction from our user makes a request for a new item, the latter is unavailable; F_1 is the probability that a repeated attempt to get the item will fail again. Now it does not matter which items the transaction is holding, only how many; the process modelling the user is essentially one-dimensional and easily solvable.

The analysis of a single user yields expressions for various characteristics in terms of F and F_1 . Three quantities are of particular interest: the average response time for a transaction, W (the time between starting and successfully completing it); the average number of items held by the user, m ; the average number of items successfully acquired by the user per unit time, λ . These three quantities are thus obtained in the form (see Chesnais, et al (2))

$$W = W(F, F_1) \tag{9.11}$$

$$m = m(F, F_1) \tag{9.12}$$

$$\lambda = \lambda(F, F_1) \tag{9.13}$$

Next, going back to the N -user model, the probabilities F and F_1 are related to the quantities already found. For instance assuming uniformity of requests over the data base, it can be argued that, since the average number of items held by all other users is $(N-1)m$ and the average number of items not held by the given user is $D-m$, the probability of failing to acquire a new item is

$$F = [(N-1)m] / (D-m) \tag{9.14}$$

A slightly more complicated, but equally straightforward argument leads to an expression for F_1 of the form

$$F_1 = F_1(\lambda, m) \tag{9.15}$$

Substituting (9.12) and (9.13) into (9.14) and (9.15) yields two fixed-point equations from which F and F_1 , and hence the other performance measures, can be determined.

For this particular system it was possible to demonstrate the existence of a solution for the fixed-point equations but not, except in some special cases, the uniqueness. However, in the examples solved numerically, the solution was always unique and provided a good approximation.

9.3.2 Local Area Networks

The three most widely known local area network architectures - Ethernet, Cambridge Ring and Token Ring - present the analyst with radi-

cally different modelling problems. In Ethernet, a station can listen to the transmission medium and, if it detects a transmission in progress, back off and try again later. A collision may occur if two or more stations start transmitting within a period less than the network propagation delay, D ; the contending parties then back off and try again later. This behaviour makes the Ethernet somewhat akin to the distributed data base of the previous sub-section, and suggests a similar approach to its analysis. A single station is considered in isolation, assuming that the influence of all other stations can be represented by fixed parameters. In the present case these are the probability that an attempted transmission will result in a collision, F , the probability that a new attempt to transmit will find the medium busy, F_1 , and the probability that a repeated attempt to transmit will find the medium busy, F_2 .

The analysis of the single station yields various performance measures such as average response time, message traffic rate, etc. - in terms of F, F_1 and F_2 . Returning to the full N -station mode network, these latter parameters are expressed in terms of performance measures already found (for example F is the probability that at least one of the other $N-1$ stations starts to transmit within an interval D each way of the instant when the given station starts its transmission; hence $F=2D(N-1)g$, where g is the rate of traffic offered by one station). This leads to the fixed-point equations

$$\begin{aligned} F &= \phi(F, F_1, F_2) \\ F_1 &= \psi(F, F_1, F_2) \\ F_2 &= \chi(F, F_1, F_2) \end{aligned}$$

As before, existence of a feasible solution can be demonstrated, but not, in general, uniqueness (Gelenbe and Mirtrani (7)).

The Cambridge Ring employs a centralised control that enforces fair sharing of the medium by a number of parallel transmissions. At the hardware protocol level, the system can be modelled by a single-server queue with a Processor-Shared scheduling strategy and state-dependant service rate (King and Mitrani (10)). That rate is a rather complex function of the number of slots circulating in the ring and the number of transmissions in progress, but once it has been derived (either analytically or empirically), performance measures are easily obtained. At the level of the Basic Block protocol, there is the additional complication that the medium can be shared only by transmissions to different destinations; the ones to the same destination are queued. The flow of messages under the Basic Block protocol is illustrated in Fig. 9.4.

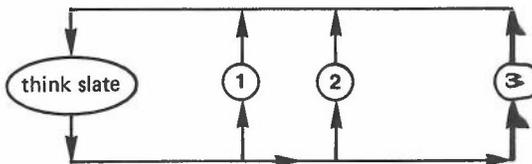


Fig. 9.4 Cambridge Ring under the Basic Block Protocol

The system can be modelled by a closed queueing network with $N+1$ nodes, numbered $0, 1, \dots, N$, and N jobs, numbered $1, 2, \dots, N$, circulating among them. If job i is at node 0 , station i is the 'think' state; if it is at node j ($j=1, \dots, N$) then station i has submitted a message addressed to station j ; that message is either waiting or is being transmitted. The network parameters are the average think times, the average transmission times at nodes $1, 2, \dots, N$ and the matrix of probabilities that job i , upon leaving node, 0 , will go to node j .

The network has a feature which prevents it from having a product-form solution: the rate at which several messages are transmitted in parallel on their number, and also on the number of messages awaiting transmission. More precisely, if there are k non-empty queues and a total of m jobs waiting for service, then the k jobs that are receiving service do so at rate $\varphi(k, m)$, where φ is a given function.

Good approximate solution can be obtained by replacing the state-dependent service rate with a constant one, C , choosing the latter so as to be consistent with the average performance of the system. For a given C , the closed network is solved to yield the average number of non-empty queues, $k(C)$, and the average number of waiting jobs, $m(C)$. The value of C is then chosen so as to satisfy the fixed-point equation

$$C = \varphi[k(C), m(C)]$$

The closed network with that value of C is used to calculate the performance measures of interest.

Our last example is the Token Ring local area network. This architecture allows a station to monopolise the entire capacity of the communication medium for the duration of a transmission. The availability of the ring is signalled by a token: the transmitting station holds the token, and releases it upon completion. The free token travels along the ring in a fixed direction; the first waiting station in its path will intercept it, regardless of how long it and the others have been waiting.

This system can also be modelled by a closed queueing network with $N+1$ nodes and N jobs. The flow of jobs is similar to the one in Fig.9.3 but with the following important differences. A job in queue i indicates that station i wishes to transmit (rather than that there is a transmission addressed to station i). Thus, job i goes from node 0 (think state) to node i , remains there until completed, then back to node 0 , etc. There is a single circulating server (the token) which serves one job at a time to completion, moving from queue i to queue $(i \bmod N)+1$.

Suppose that the performance measure of interest is, as before, the vector of average response times (W_1, W_2, \dots, W_N) , where W_i is the average interval between job i leaving node 0 and returning to it. The characteristics of the different stations, i.e. the average think times, t_i , and execution times, s_i , are given ($i=1, 2, \dots, N$). Let p_{ij} be the probability that, when job i joins queue i , the server is busy at queue j ($i \neq j=1, 2, \dots, N$). Let also R_j be the expected remaining service time for the job at queue j and D_{ji} be the average time it takes the server to reach queue i after leaving queue j . Then the average response time for station i can be written as

$$W_i = \sum_{j=1}^N P_{ij}(R_j + D_{ji}) + s_i, \quad i=1, 2, \dots, N \quad (9.16)$$

A fixed-point approximation is obtained by making simplifying assumptions which allow P_{ij} , R_j and D_{ji} to be expressed in terms of W_j and the known parameters (King et al (11), Mitrani (14)). For example, the argument aimed at p_{ij} proceeds as follows: Station j goes through 'think-transmit'

cycles whose average length is $t_j + W_j$. During each such cycle, it spends, on the average, time s_j actually transmitting. Therefore, the proportion of time that the server j spends at node j is equal to $s_j / (t_j + W_j)$. Hence, assuming that the instant of arrival of job i at node j can be treated as a random observation point (this is an approximation), p_{ij} can be set to the proportion of time that the server spends at node j , given that it is not at node i :

$$p_{ij} = [s_j / (t_j + W_j)] / [1 - [s_i / (t_i + W_i)]]$$

This, and similar arguments for R_j and D_{ji} , turn (9.16) into a set of fixed-point equations of the form^j

$$W_i = G_i(W_1, W_2, \dots, W_N), \quad i=1, 2, \dots, N$$

from which the average response times can be determined.

9.4 CONCLUSION

It is clear that, in the area of distributed computing, as in many other areas, a performance evaluation exercise has much to gain from the application of probabilistic modelling methods. The models discussed in this article can be, and have been, used successfully in performance studies of real-life systems. When the complexity of the target system is such as to preclude an exact analysis of the model, we have seen that reasonable approximations can still be obtained quite easily. Indeed, the computational cost of solving fixed-point equations is so small that even if the only benefit derived is to reduce the number of parameter points at which the system is simulated, the savings can be considerable. Nor is the application of the modelling methods limited to the type of models described here. More complex systems can often be decomposed into hierarchical levels which can then be treated separately, either exactly or approximately.

REFERENCES

- (1) Bard, Y, 1981, "A Simple Approach to System Modelling", Performance Evaluation, 1, 225-248.
- (2) Chesnais, A., Gelenbe, E. and Mitrani, I., 1983 "On the Modelling of Parallel Access to Shared Data", CACM, 26, 3, 196-202.
- (3) de Souza e Silva, E., Lavenberg, S.S. and Muntz, R.R., 1983, "A Perspective on Iterative Methods for the Approximate Analysis of Closed Queueing Networks", Procs., Int. Workshop on Appl. Maths. and Perf. Models, Pisa.
- (4) Fayolle, G., King, P.J.B. and Mitrani, I., 1982, "The Solution of Certain Two-Dimensional Markov Processes", Adv. Appl. Prob., 14, 295-308.
- (5) Fayolle, G., King, P.J.B. and Mitrani, I., 1983, "On the Execution of Programs by Many Processors", Procs., 9th Int. Conf. on Model and Perf. Eval., Maryland.
- (6) Gelenbe, E. and Mitrani, I., 1980, "Analysis and Synthesis of Computer Systems", Academic Press, London.
- (7) Gelenbe, E. and Mitrani, I., 1982, "Control Policies in CSMA Local Area Networks", Procs., Sigmetrics Conf. on Model and Perf. Eval., Seattle.

- (8) Gordon, W.L. and Newell, G.F., 1967, "Closed Queueing Systems with Exponential Servers", *Opns. Res.*, 15, 2, 254-265.
- (9) Jackson, J.R., 1963, "Jabshop-like Queueing Systems", Research Rep. 81, *Manag. Sci. Project*, UCLA.
- (10) King, P.J.B. and Mitrani, I., 1982, "Modelling the Cambridge Ring", *Procs., Sigmetrics Conf. on Model. and Perf. Eval.*, Seattle.
- (11) King, P.J.B., Mitrani, I. and Plateau, B., 1983, "Modelling a Token Ring Network", Research Rep. 12, ISEM, Univ. Paris-Sud.
- (12) Kleinrock, L., 1975, "Queueing Systems", Vols. 1 & 2, Wiley, New York.
- (13) Little, J.D.C., 1961, "A Proof of the Queueing Formula $L = \lambda W$ ", *Opns. Res.*, 9, 3, 383-387.
- (14) Mitrani, I., 1983, "Fixed-Point Approximations for Distributed Systems", *Procs., Int. Workshop on Appl. Maths and Perf. Models*, Pisa.
- (15) Mitrani, I. and Avi-Itzhak, 1968, "A Many-Server Queue with Service Interruptions", *Opns. Res.*, 3, 628-638.
- (16) Mitrani, I. and King, P.J.B., 1981 "Multiprocessor Systems with Preemptive Priorities", *Performance Evaluation*, 1, 118-125.
- (17) Neuts, M.F. and Lucantoni, D.M., 1979, "A Markovian Queue with N Servers Subject to Breakdowns and Repairs", *Manag. Sci.* 25, 849-861.
- (18) Reiser, M. and Lavenberg, S.S., 1980, "Mean Value Analysis of Closed Multichain Queueing Networks", *JACM*, 27, 313-322.

Developing concurrent systems with DTL

J. W. Hughes and M. S. Powell

Despite the fact that interest in parallel languages has arisen primarily due to the availability of parallel hardware, the real motivation for parallel languages should come from the programmer's needs. High level languages have developed precisely because they provide concepts relevant to the programmer independent of machine architectures. This paper uses simple examples to show how concurrency arises naturally during the design of programs when a language which supports the stepwise decomposition of complex operations into concurrent processes is used. The examples also illustrate the use of the DTL language developed at UMIST as part of an investigation into the use of concurrency as a structuring concept in program design.

10.1 INTRODUCTION

The concept of abstraction in program specification and design originated in Wirth's Stepwise Refinement Method (14) and Dijkstra's Structured Programming (3). The distinction between procedural abstractions for encapsulating the details of algorithms, and data abstractions, to abstract from storage representation was developed by Liskov (12) and Parnas (13). Programs designed using these methods have an organised hierarchical structure, and are consequently easier to understand and reason about than unstructured programs.

These methods allow the specification of a solution to a problem only as a sequential algorithm. They lack any concepts for expressing concurrency, and force the designer to specify operations as sequential when they have no natural order dependency. Thus, they introduce constraints into the design which are not inherent in the problem being solved and force arbitrary decisions which need never be made. At the other end of the spectrum, specifications using functional languages abstract away from operational algorithms and leave the language implementation to resolve all sequencing as well as it can. At the same time, between the sequential-operational and the functional approaches, the concept of process abstraction has emerged in the work of Hoare (4), Kahn and McQueen (10) and Jackson (9). By means of process abstraction a system can be described as a

network of concurrent processes each of which proceeds sequentially and communicates with its neighbours in the network. This is an operational approach which does not impose any unnecessary sequencing, yet models typical problem domains well and also allows a functional view to be taken. Such a description is readily understood if:

- (i) the structure of the network can be understood;
- (ii) each sequential process is simple.

All previous work has shown that a system is more readily understood if it can be decomposed hierarchically into simpler independent components. Design methods using process abstraction, therefore, must allow hierarchical decomposition of both the network and the sequential processes within it.

Distributed Translation Language (DTL) is a notation developed at UMIST for the design of systems using the concept of process abstraction.

10.2 PROCESS ABSTRACTION

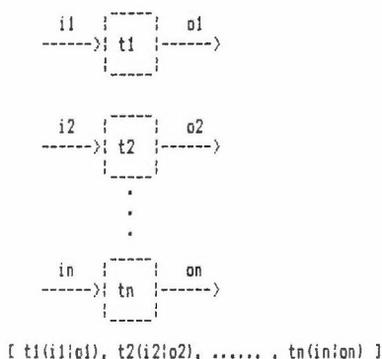
The first stage of the design process is to identify subprocesses which are required for the solution of the problem. Unless the solution consists of completely disjoint subsystems, it will usually be necessary for the subprocesses to communicate in order for them to co-operate on the solution of the problem. In most cases the subprocesses identified do not necessarily have to proceed one after the other sequentially, but may proceed concurrently, synchronised by the demands and needs of the subprocesses with which they communicate.

Concurrent situations arise primarily from two sources, one is that in which concurrency is specified in the problem (e.g. controlling a set of more than one lifts) the other arises in the decomposition itself and results in the description of some function as the composition of simpler functions. Network diagrams such as those shown in figure 10.1 give a visual description of the structure of a decomposition. The boxes represent processes and the arcs their intercommunication. The decomposition proceeds by further refining the description of each process by repeating the procedure outlined above. At each stage the nature of the data communicated on the arcs which are introduced by the decomposition is specified. These are sequential streams of different kinds of data items and may be described by grammars.

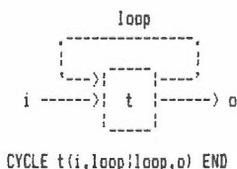
Eventually the refinement reaches a level where the process required to input the specified data streams and produce the required output streams is naturally sequential. DTL is therefore designed to allow specification of:-

- i) The hierarchical structure of networks of communicating processes.

iii) A disjoint parallel composition of n simpler translations.



iv) A cyclic composition of ii) or iii).



A concurrent DTL translation is thus specified by a network expression, formed using three operators (corresponding to the three forms of construction) with translations as operands.

The expressions described so far lead to finite cyclic networks of translations. However, it is possible to use the translation being defined in its defining expression, recursively.

10.4 EXAMPLE : SORTING

An illustration of hierarchical process refinement is the problem of sorting a sequence of data into descending order. Suppose the input is a stream of integers terminated by an end marker. The sorting process can be refined into two processes, first one which finds the maximum value and separates it from the rest; then one which sorts the rest and outputs the maximum followed by the sorted remainder. This last process can be refined into two processes, one to sort the rest and the other to append the result to the maximum. These two refinements are shown in figure 10.1.

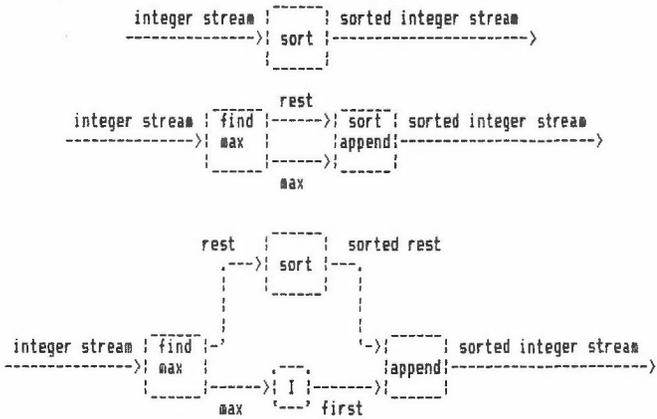
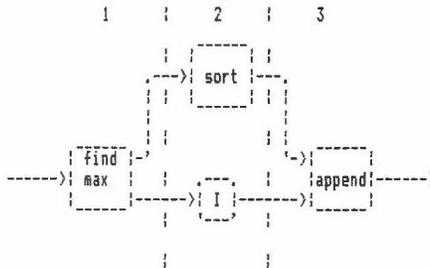


Fig.10.1

The network in figure 10.1 consists of a three stage pipeline as shown in figure 10.2 with the corresponding DTL network expression.



```

TRANSLATION sort(in ! out) =
  findmax(in!max,rest)
  >>
  [ sort(rest!sortedrest), I(max!first) ]
  >>
  append(first,sortedrest!out)
END

```

Fig.10.2

The translation I is the identity translation. It is analogous to the empty statement in a Pascal-like language

in that it allows any network to be expressed in a well structured fashion.

The next stage in the refinement requires the specification of `findmax` and `append`. These are most naturally expressed as simple sequential translations.

10.5 SEQUENTIAL TRANSLATIONS

A sequential translation translates its input streams into its output streams. As mentioned earlier, the streams are sequences of data items which may be of different kinds. A stream is defined in terms of the kinds of data item which it may communicate (its alphabet), and the permissible sequences in which those items may occur. Any data item kind may have associated attributes. e.g. an integer type item may have an attribute representing its value. The structure of the sequence can be described by a grammar.

The structure of the sequential process which consumes the inputs and generates the outputs naturally reflects the structure of its streams. Thus a sequential translation is derived from the grammars of its input and output streams using much the same principles as the Jackson program design method (9) or recursive descent compilers (1). A DTL sequential translation therefore specifies its stream alphabets and a set of production rules in extended BNF defining the translation grammar. Nonterminals in the productions enforce a hierarchical structure.

In the previous sort example the final process in the pipeline is a simple sequential translation. It has two input streams, the first communicates the maximum value, or an end marker if the input sequence contains no integers, the other the sorted remainder of the integers or nothing at all. The output stream from `append` is to communicate a (possibly empty) sequence of integers in descending order followed by an end marker. These stream structures can be described by grammars as follows:-

```

<first> ::= int(n) | endstream
<rest>  ::= *( int(x) ) endstream | <empty>
<out>  ::= *( int(y) ) endstream

```

The `append` function is clearly achievable by forming the stream `out` from `first` with `rest` appended to it, which can be thought of as requiring

```

<out> = <first> <rest>

```

to be invariantly true for all executions of the translation `append`. This property can be used to derive the `append` translation. By considering the grammar for `first`, two cases have to be considered:-

1) `<first> = endstream`

It is intended that in this case the `rest` stream will be empty and the `out` stream will be the same as the `first` stream,

this is described by the following production

```
first.endstream [out.endstream]
```

ii) $\langle \text{first} \rangle = \text{int}(n)$

In this case the `int` should be the first symbol in `out` and the rest of `out` should consist of the rest stream, this is described by

```
first.int(n) [out.int(n)]
*( rest.int(i) [out.int(i)] )
rest.endstream [out.endstream]
```

The `append` translation consists of just these two alternatives and is therefore given by the sequential translation in figure 10.3.

```
TRANSLATION append(first, rest ; out)
  first, rest, out = int(n: integer), endstream
  <append> ::= first.int(x) [out.int(x)]
             *( rest.int(x) [out.int(x)] )
             rest.endstream [out.endstream]
             ;
             first.endstream [out.endstream]
END
```

Fig.10.3

Data items and productions may have attributes. This facility is used in the definition of `findmax` shown in figure 10.4.

```
TRANSLATION findmax(in ; max, rest)
  in, rest, max = int(n: integer), endstream
  <findmax> ::= in.int(maxvalue) <body(maxvalue)>
              in.endstream [max.int(maxvalue)] [rest.endstream]
              ;
              in.endstream [max.endstream];
  <body(maxvalue: integer)> ::=
    *( in.int(x: x <= maxvalue) [rest.int(x)]
      ;
      in.int(x: x > maxvalue) [rest.int(maxvalue)]
      <maxvalue:=x>
    )
END
```

Fig.10.4

The first item accepted by `findmax` is either an integer or an end marker if the input sequence is empty. In the latter case, the end marker is output on the stream called `max` and the translation is complete. Otherwise the value of the attribute of the first integer is assigned to the variable `maxvalue`. The rest of the integers on the stream called `in` are then processed by the production named `body` until an end marker is encountered. An integer is then transmitted on the stream called `max` with an attribute value corresponding to the value of the variable `maxvalue`. This is followed by transmitting an end marker on the stream called

rest and the translation is again complete.

The production `body` processes the remainder of the (zero or more) integers from the stream called `in`. All those with attribute values less than or equal to the current value of `maxvalue` are transmitted on the stream called `rest`. Whenever an integer is accepted with an attribute value greater than the current value of `maxvalue`, `maxvalue` is transmitted on `rest` and then redefined to have the value of the last integer accepted on `in`. The initial and final values of `maxvalue` are communicated to and from the root production via the attribute of `body`.

Sequential translations thus have a hierarchical structure supported by production names. The form of production definitions supports the conventional control structures of sequence, selection and iteration. The notation also enforces a correspondence between input and output data structures and control structures as advocated by Jackson (9). Furthermore the concurrent facilities in DTL, in allowing natural concurrency to be retained in a program description, tend to lead to sequential translations which are small and simple to understand. Many simple sequential translations like `append` in the above example could be provided as standard components by an implementation of the language.

10.6 EXAMPLE : LOGIC NETWORKS

In this section the use of DTL as a hardware description language will be considered. Hardware description languages allow logic networks to be described and simulated from a description of how a set of primitive components are connected together to form a complete hardware system. The primitive components are generally logic gates. e.g. NAND gates, NOR gates etc. These may be connected together in a modular fashion and a hardware description language should allow this modularity to be expressed. e.g. Two NAND gates may be connected together to form a flip-flop and it should be possible to describe a 16-bit register made from 16 flip-flops without having to describe the internal structure of each one.

In order to use DTL as a hardware description language the nature of the signals which are communicated between logic gates must be defined and then a set of primitive gates can be designed. One of the advantages of using a general purpose language for hardware description in this way, is that the primitive elements can be described in as much detail as is required for any particular application. Hardware description languages in general, supply a fixed set of primitives which may not be changed by the ordinary user.

Hardware engineers often abstract away from the concurrent nature of logic networks and consider logic gates to be represented by pure Boolean functions. In this style an AND gate would be represented by a function with the

following type.

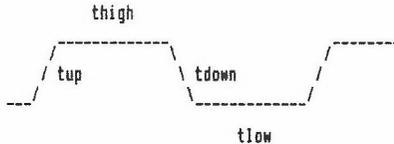
```
and: Boolean x Boolean -> Boolean
```

However for the purposes of continuous simulation it is important to regard a logic network as set of communicating processes connected together by streams of Boolean values. In this style an AND gate must be represented by a function with the following type.

```
and: Boolean* x Boolean* -> Boolean*
```

A description of this kind may enable the synchronisation of events in a logic network to be reasoned about but still abstracts away from many of the factors which complicate logic design and lead to 'bugs' in hardware systems. e.g. Gate delays and edge speeds. If a hardware description language is to be useful it must enable such factors to be represented so that they can either be reasoned about during the design process and the behaviour of a complete system can then be proved to meet its specification, or the language description can be used as the basis for accurate simulation and attempts can be made to establish the correct behaviour of a system by testing before the real hardware is built.

The diagram below shows an example of the kind of signal which might be communicated between two gates in a logic network. It illustrates some of the factors which a hardware description language should take into account.



In this diagram *thigh* and *tlow* represent the time periods which the signal spends in the high and low states and *tup* and *tdown* represent the transition times between the high and the low states during which its state is undefined. This representation abstracts away from the absolute values of voltage, current, pressure or magnetic flux density etc, which may be used to implement the high or low signal levels. In practice this is a reasonable thing to do due to the nature of the hardware support provided for these concepts by physical logic gates. If such signals are to be modelled by DTL streams, their alphabet may be described as shown below.

```
signal alphabet = up(time: natural), down(time: natural), high(time: natural), low(time: natural)
```

Many classes of signal can be modelled using this alphabet in terms of the transitions they contain and the stable states in between. Some examples are shown below.

```

<square wave(period: natural)> ::= low(period) up(edge time) high(period) down(edge time)
<pulse(length: natural)> ::= up(edge time) high(length) down(edge time)
<pulses(interval, length: natural)> ::= low(interval) <pulse(length)> <pulses(interval, length)>
<glitch> ::= up(edge time DIV 2) down(edge time DIV 2)
<noise> ::= low(random period) <glitch> <noise>

```

It may be that the environment in which a particular hardware system is to be implemented is so well behaved that glitches and noise do not occur sufficiently frequently to cause problems. It may also be the case that the edge times involved are short compared to the stable high and low states of the signals and that it is the synchronisation of transitions between states which is of interest rather than the absolute time periods which they spend in high or low states. If (fortuitously for the complexity of the examples which follow) all these things are the case, the alphabet required to describe all possible signals can be reduced to that shown below.

```
simple signal alphabet = up, down
```

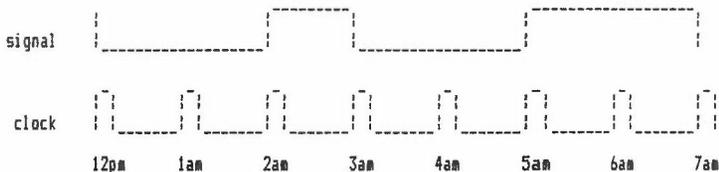
In such a wonderfully simple universe of signals, there are only three classes of signal. There are empty signals which contain no transitions and carry no information. There are signals which start life by making a transition from a low state to a high state and there after go down, up, down, etc and there are signals which start life by making a transition from a high state to a low state and there after go up, down, up, etc.

```

<signal> ::= <empty> | <low> | <high>
<low> ::= up (<high> | <empty>)
<high> ::= down (<low> | <empty>)

```

If we require some timing information about such signals we can compare them with a special clock signal which is known to make transitions at regular intervals. The specification of such a clock signal is outside the scope of this paper as it may require some non-digital components in its implementation!



DTL translations may be constructed which process such signals and produce new signals as their outputs. A primitive logic gate such as a NOT gate can be described as shown below.

```

TRANSLATION not(in ! out)
  in , out = up, down
  <not> ::= <empty> ! <low> ! <high>
  <low>  ::= in.up out.down (<high> ! <empty>)
  <high> ::= in.down out.up ( <low> ! <empty>)
END

```

The above translation may be constructed systematically by using a Jackson-like technique and has the advantage of making it easy to establish that inputs in the form of signals are accepted, that the output is always a signal and that it represents the 'not' of the input. In an environment where the input is always guaranteed to be a signal, simple program transformation techniques can be used to derive the less complex translation which follows.

```

TRANSLATION not(in ! out)
  in , out = up, down
  <not> ::= <empty> ! in.up out.down <not> ! in.down out.up <not>
END

```

More complex gates may be described in a similar fashion by sequential DTL translations. These may then be connected together into still more complex structures by using the DTL network composition operators. The example below shows how a NAND gate may be constructed from a NOT gate and an AND gate.

```

TRANSLATION nand(in1, in2 ! out) = and(in1, in2 ! x) >> not(x ! out)

```

As is well known two NAND gates may be connected together to make a primitive memory element in the form of a reset/set flip-flop. The same is of course true of the NAND gates described above.

```

TRANSLATION flipflop(in1, in2 ! out1, out2) =
  CYCLE
  [ nand(in1, out2 ! out1), nand(in2, out1 ! out2) ]
  END
END

```

For simulation purposes it would generally be more efficient to use a single sequential translation to describe the same function, however there is an obvious advantage to being able to describe such a component in as much detail as is required. Many hardware description languages are forced to provide such components as standard parts of the language because they cannot be constructed from the primitives of the language.

The last example in this section shows how an n-bit register may be constructed from n flip-flops by using an array of translations. DTL supports arrays of translations and arrays of streams.

```

TRANSLATION register(in1[1..n], in2[1..n] ; out1[1..n], out2[1..n]) =
    flipflop[i:1..n](in1[i], in2[i] ; out1[i], out2[i])
END

```

The network expression in the example above can be regarded as a shorthand for

```

[ flipflop[1](in1[1], in2[1] ; out1[1], out2[1]),
  :
  :
  flipflop[n](in1[n], in2[n] ; out1[n], out2[n])
]

```

The use of DTL as a hardware description language has been illustrated in the preceding examples by developing descriptions for common hardware components in a bottom-up fashion. For hardware systems of any complexity the normal process of design would proceed top-down using the structuring methods provided by the language to support stepwise refinement. The leaves of a refinement tree which describes this process would represent the kind of primitive hardware components described above.

The set of such components can be defined in advance for a specific implementation environment. It might, for example, comprise a subset of the 74 series TTL gates. This situation is very similar to the use of top-down stepwise refinement in the production of software, where the leaves of refinement trees represent high-level language statements or, perhaps if a low level language is used, individual machine instructions.

In all of the preceding examples, the signals communicated between gates have had a very simple form and only record the transitions between logic states. Where more information must be represented because, for example, edge speeds are important in the implementation environment, the structure of signals may be augmented with such information and the primitive gate translations updated accordingly. The edge speed of an output transition may be determined by the edge speed of the input transition which caused it. Edge triggered flip-flops may be described which will only respond to suitably fast edges. By augmenting the information which defines a signal, all of the necessary imperfections of physical hardware devices may be modelled.

10.7 IMPLEMENTATION OF DTL PROGRAMS

The design of DTL has been motivated by the programmer's requirements for expressing algorithmic solutions to problems. The language may be implemented on a variety of machine architectures. On a single processor, multiplexing between the individual translations can be achieved using a coroutine mechanism. On multiprocessor architectures, the individual translations would be distributed amongst the available processors on either a

one-to-one basis or by again employing a coroutine mechanism to multiplex a number of translations on each processor. In this case, streams may be implemented either in a shared store on closely coupled architectures or by direct processor to processor communication on loosely coupled systems. A detailed description of a virtual machine architecture for supporting DTL programs is given in (7) The above implementation possibilities are all special cases of the virtual machine architecture described.

10.8 SUMMARY

DTL has been designed by analysing the programmer's requirements and is based on a data driven approach to program design. An overview only has been given here. A full language description is given in (6). The resulting language allows programs to be expressed as structured networks of translations which communicate data on fully synchronised streams. The network structure allows natural concurrency in the problem to be maintained in the solution, without the introduction of any unnatural sequencing. Reasoning about the behaviour of DTL programs is facilitated by the hierarchical structure, the absence of shared data and the full synchronisation.

The language has been used for a variety of applications, only two of which have been illustrated here. It has proved suitable for describing conventionally concurrent systems such as a spooler, less obviously concurrent algorithms such as sorting, and less obviously algorithmic objects such as logic networks.

The language may be readily implemented on a variety of computer architectures. In the case of multiprocessor architectures, design in DTL results in programs which, because they maintain all the concurrency inherent in the problem, execute with enhanced performance. For example, if sufficient processors are available, the sort program described in this paper will run in linear time. Thus the most efficient program from the execution viewpoint is also the most efficient from the design viewpoint because it reflects the natural structure of the problem. There is therefore no need to distort the natural solution in order to optimise it.

REFERENCES

1. Aho A & Ullman J
"The Theory of Parsing, Translation and Compiling:
Volume 1, Parsing"
Prentice Hall, (1972)
2. Coleman D, Hughes J W & Powell M S
"A Method for the Syntax Directed Design of
Multiprograms"
IEEE Transactions on Software Engineering, SE-7, No. 2,
pp 189 - 196, (1981)

3. Dijkstra E W
"Notes on Structured Programming"
APIC Studies on Data Processing, No. 8, Academic Press,
pp 1-81, New York, (1972)
4. Hoare C A R
"Communicating Sequential Processes"
CACM, Vol 21, No.8, pp 666-677 (August 1978)
5. Hughes J W & Powell M S
"Program Specification Using DTL"
Workshop on Program Specification, Aarhus, (1981)
6. Hughes J W & Powell M S
"DTL: A Language for the Design and Implementation of
Concurrent Programs as Structured Networks"
Software - Practice and Experience, Vol. 13, No. 12,
pp 1099-1112 (1983)
7. Hughes J.W. and Powell M.S.
"The Implementation of DTL"
Software - Practice and Experience, Vol. 13, No. 12,
pp 1113-1128 (1983)
8. Hughes J W "A Formalisation and Explication of the
Michael Jackson Method of Program Design"
Software - Practice and Experience, Vol.9, No.3,
pp 191-202, (1979)
9. Jackson M A
"Information Systems : Modelling, Sequencing and
Transformations" IEEE Proc. International Conference on
Software Engineering, (1978)
10. Kahn G & McQueen D B
"Coroutines and Networks of Parallel Processes"
IFIP Conference on Information Processing, (1977)
11. Lewis P M & Stearns R E
"Syntax Directed Transductions"
JACM, Vol.15, No.3, pp 465-488(July 1968)
12. Liskov B
"Modular Program Construction Using Abstractions"
Lecture Notes in Computer Science, No. 86,
Springer Verlag, pp 354-389 (1980)
13. Parnas D L
"Information Distribution Aspects of Design Methodology"
Information Processing 71, Vol.1, North Holland
Publishing Co., pp 339-344 (1972)
14. Wirth N
"Program Development by Stepwise Refinement"
Communications of the ACM, Vol. 14, No.1 (April 1971).

ACKNOWLEDGEMENTS

The work described was partially supported by SERC DCS research grants no. GR/A74678 and GR/B35062. The authors would also like to thank Chris Tan for his suggestions and contributions to the work described.

Chapter 11

Parallel algorithm design

D. J. Evans

11.1 INTRODUCTION

A description of the work on distributed and parallel processing systems involving parallel algorithms design undertaken in the Department of Computer Studies, Loughborough University of Technology, is given.

The work is based on production minicomputers connected by a shared memory to achieve certain levels of distributed and parallel processing. By the use of the manufacturer's software and a number of software modifications a parallel system can be set up and be made operational in a matter of months not years. This strategy has resulted in the 2 operational Loughborough M.I.M.D. systems:

1. Dual Interdata 70 system, 1976-79;
2. 4 Texas 990/10 NEPTUNE system, 1979-1984.

These parallel systems have provided a service in the Department and to other SERC users for the past 5 years in which research into parallel algorithms have been extensively undertaken.

Parallelism arises at many different levels within a complex problem which if exposed can be efficiently exploited. By incorporating software tools in the system to measure the performance we are able to restructure our algorithms or component parts of them into parallel form to run more efficiently.

In particular, we have studied the various standard techniques of achieving parallelism, i.e. vectorization, problem partitioning and divide and conquer strategies as well as exploiting the use of implicit parallelism in various numerical and non-numerical algorithms. In addition, new parallel algorithms have been introduced.

The principles we have learnt from this study have also been extended to algorithms on other parallel systems, i.e. DAP and CRAY.

11.2 NUMERICAL PARALLEL ALGORITHMS

The main aims of the work have been as follows:

1. To discover and design alternative solution methods which offer parallelism in one form or another.
2. To study the suitability of each parallel scheme for implementation on different parallel computer systems.
3. To obtain the performance analysis of the implemented procedures.

The primary feature that distinguishes parallel algorithms and systems from the more usual uniprocessor situation is that parallelism entails the use of facilities or resources not present in sequential solutions,

for different values of λ_1 . The number of negative q_i 's, $i=1,2,\dots,n$ of the above sequences will indicate the number of eigenvalues below the sample point λ_1 . Repeated application of this procedure will separate the eigenvalue spectrum into small sub-intervals of size ϵ (pre-defined) which contain 1 or more eigenvalues, λ . In the following section we shall briefly discuss the possibilities for solving this problem on alternative parallel systems.

11.2.2 Parallel Methods for the Tridiagonal Eigenvalue Problem

The standard procedure to solve the above problem on sequential computers involves halving the interval containing all the eigenvalues and successively choosing one of the two sub-intervals containing eigenvalues. This is in turn bisected into two further sub-intervals and the process is continued until the eigenvalues are separated to a predefined accuracy. The method of *parallel bisection* (Barlow and Evans, [1]) uses the principle on all the previously determined non-empty sub-intervals. A major disadvantage with this scheme is that the number of parallel processes available at the initial stages of the algorithm is limited to 1 at the 1st iteration, 2 in the 2nd iteration, 4 in the 3rd iteration and so on. Therefore, the potential speed up obtainable from this parallel version is dependent upon the number of non-empty intervals available at each stage and is bounded by N , the maximum number of eigenvalues.

However, in the multisection procedure, each subinterval is divided into m (instead of 2 as in the bisection procedure) subintervals for each of which the Sturm sequence is evaluated in parallel on each of the available processors simultaneously. This procedure can be extended further into a method called *parallel multisection* (Barlow et al [2]) in which the above two methods are combined to obtain greater efficiency. In this method given p processors and m domains then one allocates $\ell=p/m$ processors per domain. The speed-up of the resultant method lies between m and $m \log_2(\ell+1)$.

In all of the above methods the recurrence relation is evaluated sequentially and parallelism is generated through the simultaneous evaluation of several sequences for different sample points. However, additional parallelism through the reformulation of the Sturm sequence itself can be achieved as the following two algorithms will show.

The method of *recursive doubling*, (Lambiotte, [3]), for example, re-defines the Sturm sequence as:

$$\begin{aligned}
 p_0 &= 1, \quad p_1 = c_1 - \lambda_1, \\
 p_i &= (c_i - \lambda_1)p_{i-1} - b_i^2 p_{i-2}, \quad i=2,3,\dots
 \end{aligned}
 \tag{2.2}$$

which can be expressed in the form,

$$\begin{bmatrix} p_i \\ p_{i-1} \end{bmatrix} = \begin{bmatrix} i \\ \prod_{j=1}^i s_j \end{bmatrix} \begin{bmatrix} p_1 \\ p_0 \end{bmatrix}, \quad i=2,3,\dots,n
 \tag{2.3}$$

where,

$$s_j = \begin{bmatrix} c_j - \lambda_1 & -b_j^2 \\ 1 & 0 \end{bmatrix}.
 \tag{2.4}$$

It can be easily seen that the sequences p_i and q_i , $i=1,2,\dots,n$ are related through the formula,

$$q_i = \frac{P_i}{P_{i-1}} \quad (2.5)$$

The parallelism in the above procedure is obtained by evaluating the (2*2) matrix multiplication terms $\prod_{j=1}^i S_j$, $i=2,3,\dots,n$ in the following manner,

processor				
1	S_1			
2	$S_2 * S_1$			
3	$S_3 * S_2 * S_1$			
4	$S_4 * S_3 * S_1 S_2$			
5	$S_5 * S_4 * S_2 S_3 * S_1$			(2.6)
6	$S_6 * S_5 * S_3 S_4 * S_1 S_2$			
7	$S_7 * S_6 * S_4 S_5 * S_1 S_2 S_3$			
8	$S_8 * S_7 * S_5 S_6 * S_1 S_2 S_3 S_4$			
9	\vdots	\vdots	\vdots	\vdots
\vdots	\vdots	\vdots	\vdots	\vdots

It can be seen from (2.3) that all the terms are built up of products of the S_j matrices. The system can be solved in $\log_2(n)$ sequential stages, each stage consisting of between $n/2$ and n parallel processes each of which is a multiplication of (2*2) matrices. Thus, stage 1 forms all products $(S_j \cdot S_{j-1})$ for $j=2,3,\dots,n$, stage 2 combines all the results to give the products $(S_j \cdot S_{j-1}) \cdot (S_{j-2} \cdot S_{j-3})$ for $j=4,5,\dots,n$ and so on.

Thus, the parallel evaluation of the Sturm sequence can be completed in a time

$$T_r = (12n/p)\log_2(n) + 3n/p$$

using p processors and taking into account the cost of counting the sign changes in the p_i sequence.

Finally, we look at an alternative procedure which reformulates the process of the evaluation of the recurrence relation in order to generate further parallelism.

The *Block Triangular method* due to Chen et al, [4] was originally designed for the solution of banded unit lower triangular systems of equations with bandwidth $m+1$ where m and the number of available processors p are assumed to be smaller than the size of the system n .

It is well known that any linear recurrence relation can be expressed as a banded unit lower triangular system which can then be decoupled into a number of parallel processes using $\log_2(n) < p < n$ processors. In particular, the Sturm sequence (2.2) can be rewritten as a unit lower triangular system of equations $Lp = f$ with the coefficient matrix L of bandwidth 3, with elements $l_{i,j}$ such that

$$l_{i,j} = \begin{cases} 1 & , i=j \\ -(c_i - \lambda_1) & , i=j+1 \\ b_i^2 & , i=j+2 \\ 0 & , \text{Otherwise} \end{cases} \quad (2.7)$$

174 Parallel algorithm design

be efficient relative to the best sequential algorithm for that problem. Therefore, there is little use if the sequential starting point is inefficient or the cost of reformulation of a sequential procedure into a parallel version is too high. For example, the cost of evaluation of the Sturm sequence sequentially is $3n$ floating point operations (flops) where n is the size of the system. The recursive doubling procedure for evaluating the same sequence costs $12n$ flops. Therefore, it can be seen that the introduction of parallelism within the evaluation of the recurrence relation has produced a procedure which is immediately more expensive by a factor of 4. Thus, since the maximum speed up of the recursive doubling procedure is only $\log_2 p$ the actual speed up compared with the sequential bisection is $\log_2 p/4^2$ if the processors on the parallel system have equal computing power to that of the sequential system.

To compare the performance of the above alternative solution methods we analyse their individual computing times for evaluating a single eigenvalue as follows:

sequential bisection	$T = 3n$
parallel bisection ($N > p$)	$T_D = 3n/p$
parallel multisection	$T_m = 3n/\log_2(p+1)$
recursive doubling	$T_r = (12n/p)\log_2(n) + 3n/p$
block triangular method	$T_B = 15n/p + 12\log_2(p)$

When trying to locate large numbers of eigenvalues ($N > p$) the parallel bisection method is the best solution. However, for a single eigenvalue then considering the above computational costs it can be seen that after some judicious reasoning the following conclusions for the best method can be made, i.e.,

$$T_r < T_m \text{ if } p/\log_2 p > 4\log_2(n) .$$

Thus, for $n=1024$, for example, the recursive doubling strategy is better than multisection if the number of processors is greater than 512 which signifies that it is suitable for the DAP but not for the CRAY or NEPTUNE.

Furthermore, $T_B < T_m$ if $15n/p + 12\log_2 p < 3n/\log_2(p+1)$ which indicates that p has to be greater than 28 signifying the suitability of the block triangular method for the CRAY and the DAP but not for NEPTUNE.

Finally, $T_m < T_B$ if $\log_2(p+1) > n/4 \log_2 n$ with $p \leq n$. For $n=1024$, this means that multisection is better than the block triangular method if $p > 32,000,000$ which indicates that it is unsuitable for use on the CRAY, DAP and the NEPTUNE system.

Another important criteria in designing effective parallel algorithms is the stability of the proposed solution methods. Unstable solutions are of little interest to the user and current sequential algorithms have had their stability properties thoroughly analysed. Therefore, any parallel solution that deviates from the calculation of a sequential solution must be thoroughly analysed for stability.

It can be easily seen that the computations involved in the parallel bisection and multisection algorithms follow that of the sequential bisection method. However, in the recursive doubling and block triangular methods the computation involves matrix multiplications instead of scalar operations as in the previous procedures. There still remains further work to be done, to verify the stability characteristics of these procedures. For example, in the recursive doubling procedure to produce

results of identical accuracy to that obtained by the standard bisection method all the elements involved in the matrix operations have to be stored in double precision. This in turn will affect the performance of the algorithm on systems such as DAP where the performance of the system as a whole degrades as the word length increases (e.g. on the DAP, the timing for the multiplication of two numbers stored as R*4 is 274 µseconds whereas for R*8 it is 1066 µseconds).

11.2.4 Results

Combined multisection and bisection has been successfully implemented on 2 different types of parallel architectures. These are the ICL-DAP, an array processor and the CRAY-1S vector processor. The recursive doubling method has also been implemented on two systems, whilst the block method is currently being implemented (Barlow et al [6]).

The implementations are fully described by Barlow et al, [2]. The results are summarised in Tables 2.1 and 2.2.

TABLE 2.1 ICL DAP and CRAY-1 timings for locating all the eigenvalues using combined multi-section and bisection

SIZE	ICL-DAP		CRAY-1	
	Time (secs.)	Speedup*	Time (secs.)	Speedup***
64	0.24	4	0.028	3.8
256	1.15	12	0.27	5.5
1024	6.66	27	3.14	6.2
4096	65.15	>46**	49.26	6.8

* speed-up calculated with respect to ICL 2980 (the DAP host)

** ICL 2980 version ran out of time

*** compared to CRAY sequential solution

TABLE 2.2 ICL-DAP and CRAY-1 timings for locating a small number of eigenvalues of a matrix of size 1024

No. of Eigen- values	ICL-DAP		CRAY-1	
	Multi- section & bi- section	Recursive Doubling	Multi- section & bi- section	Recursive Doubling
1	2.85 secs	1.1 secs	0.034 secs	0.145 secs
4	2.85	2.29	0.0485	0.312
16	2.85	6.8	0.0929	1.128

N.B. the coefficient matrix for the above table is the tridiagonal matrix A defined as

$$a_{ij} = \begin{cases} 2 & \text{if } |i-j|=0 \\ -1 & \text{if } |i-j|=1, \quad i, j=1, 2, \dots, n \\ 0 & \text{otherwise} \end{cases}$$

required. In the second step, $1/4(N/M)$ merges are required to sort the subsets of size 2 elements each which are the results of the first step. Thus, on average, $3/4(N/M)$ comparisons are required. This proceeds until the final step where only one merge is completed to merge two

subsets of size $2^{\log(N/M)-1}$ each. Hence, the total number of comparisons for this step are

$$(2 \times 2^{\log(N/M)-1} - 1) = 2^{\log(N/M)} - 1.$$

The sum of the comparisons of the $\log(N/M)$ steps gives the total complexity of one subset of size (N/M) . Hence, the complexity of the M subsets is given by

$$\begin{aligned} C_{1s} &= M \left[\frac{1}{2} \left(\frac{N}{M} \right) + \frac{3}{4} \left(\frac{N}{M} \right) + \dots + \frac{2^{\log(N/M)-1}}{\log(N/M)} \left(\frac{N}{M} \right) \right] \\ &= N \left[1 - \frac{1}{2} + 1 - \frac{1}{4} + \dots + 1 - \frac{1}{2^{\log(N/M)}} \right] \\ &= N \left(\log \left(\frac{N}{M} \right) - 1 \right) + M \text{ comparisons.} \end{aligned} \tag{3.1}$$

The M subsets can be sorted in parallel where M parallel paths are generated with the condition that $M \geq P$, where P is the number of processors. In this case, at most $\lceil M/P \rceil$ parallel paths are performed by each processor where each path requires the same complexity described above. Therefore, the total complexity of this algorithm in the parallel implementation is

$$C_{ps} \leq \frac{N}{P} \left(\log \left(\frac{N}{M} \right) - 1 \right) + \frac{M}{P} + 1 \text{ comparisons.} \tag{3.2}$$

As the speed-up, S_{ps} , ratio is important in most of the parallel implementations, therefore,

$$S_{ps} = \frac{C_{1s}}{C_{ps}} \approx O(P).$$

This means that the speed-up of this algorithm is always linear (i.e. of $O(P)$) and does not depend on M .

The complexity of the 2-way merge algorithm that merges the M subsets in $\log M$ steps is measured as follows:

In the first step, $M/2$ paths are generated where each path merges two neighbouring subsets of size N/M each. This results in $(M/2)(2N/M - 1)$ comparisons. Similarly, in the second step, each two subsets of size $(2N/M)$ are merged in which case $M/4$ paths are required, and this yields $(M/4)(4N/M - 1)$ comparisons. The algorithm proceeds until the final step in which only $M/2^{\log M - 1}$ path is generated to merge two subsets of size $\frac{2^{\log M - 1}}{M}$. This gives $\frac{M}{2^{\log M}} \left(2 \cdot \frac{2^{\log M - 1} \cdot N}{M} - 1 \right)$ comparisons as the complexity of this step.

By summing up the complexities of all the steps we obtain the total complexity in one processor t_{1M} . Thus,

$$t_{1M} = N \log M - \frac{M}{2} \cdot 2 \left(1 - \frac{1}{2^{\log M}} \right) = N \log M - M + 1. \tag{3.3}$$

In the parallel implementation, if $M \geq P$, then each processor has to

carry out one or more paths in the first few steps. Precisely, when $M \geq 2^i \cdot P$, for $i=1,2,\dots$, all the co-operating processors are potentially activated so that no loss in the algorithm's efficiency is achieved. In other words, when the step number i such that $i \leq \log(M/P)$, for $i=1,2,\dots$, all the processors are potentially used. After these $\log(M/P)$ steps, the number of processors is halved in each step until the final step where only one processor is active while $(P-1)$ processors remain idle. The number of steps between $\log(M/P)$ and the final step are $\log(P)$, where $\log(M/P) + \log P = \log M$ steps represent the total number of steps to merge the M subsets.

Now, by proceeding as in the sequential case with the addition of the above-mentioned properties, the total complexity of the 2-way merge algorithm when run in parallel becomes:

$$\begin{aligned}
 t_{PM} = & \left\lceil \frac{M}{2P} \right\rceil \left(\frac{2N}{M} - 1 \right) + \left\lceil \frac{M}{4P} \right\rceil \left(\frac{4N}{M} - 1 \right) + \dots + \left\lceil \frac{M}{2^{\log(M/P)} \cdot P} \right\rceil \\
 & \left(\frac{2^{\log(M/P)} \cdot N}{M} - 1 \right) + \left\lceil \frac{M}{2^{\log(M/P)+1} \cdot P/2} \right\rceil \left(\frac{2^{\log(M/P)+1} \cdot N}{M} - 1 \right) + \dots + \\
 & \left\lceil \frac{M}{2^{\log M} \cdot P} \right\rceil \left(\frac{2^{\log M} \cdot N}{M} - 1 \right) . \quad (3.4)
 \end{aligned}$$

Equation (3.4) can be simplified with the use of the rules of the geometric series. Thus,

$$t_{PM} \leq \frac{N}{P} \log \left(\frac{M}{P} \right) + \frac{2N}{P} (P-1) - \frac{M}{P} + 1 + \log \left(\frac{M}{P} \right) . \quad (3.5)$$

Hence, the merge speed-up, S_{PM} , which equals $\frac{t_{1M}}{t_{PM}}$ becomes of

$$O\left(\frac{P \log M}{\log \left(\frac{M}{P} \right) + 2P - 2} \right) .$$

If we sum the results of equations (3.1) and (3.3) and equations (3.2) and (3.5), we obtain respectively, the total sequential and parallel complexities of the neighbour sort with the 2-way merge algorithm. Thus,

$$\begin{aligned}
 T_1 &= N \left(\log \left(\frac{N}{M} \right) - 1 \right) + M + N \log M - M + 1 \\
 &= N (\log N - 1) + 1 , \quad (3.6)
 \end{aligned}$$

and,

$$\begin{aligned}
 T_P &= \frac{N}{P} \left(\log \left(\frac{N}{M} \right) - 1 \right) + \frac{M}{P} + 1 + \frac{N}{P} \log \left(\frac{M}{P} \right) + \frac{2N}{P} (P-1) - \frac{M}{P} + 1 + \log \left(\frac{M}{P} \right) \\
 &\leq \frac{N}{P} \left(\log \left(\frac{N}{P} \right) - 1 \right) + \frac{2N}{P} (P-1) + \log \left(\frac{M}{P} \right) + 2 . \quad (3.7)
 \end{aligned}$$

Then, the total speed-up, S_{Pt} , can be obtained. Thus,

$$S_{Pt} \geq \frac{N (\log N - 1) + 1}{\frac{N}{P} \left(\log \left(\frac{N}{P} \right) - 1 \right) + \frac{2N}{P} (P-1) + \log \left(\frac{M}{P} \right) + 2} . \quad (3.8)$$

After simplification S_{Pt} becomes $O\left(\frac{P \log N - 1}{\log \left(\frac{N}{P} \right) + 2P - 2} \right)$. From equation

(3.8), we notice that the total speed-up is mainly dependent upon the number of processors P and independent of the number of paths M . The experimental and theoretical values of the speed-up ratios are presented respectively in Tables 3.1 and 3.2, where the data since N is

1024.

TABLE 3.1: The experimental results of the algorithm

No. of Processors (P)	No. of Paths (M)	Sorting Speed-up	Merging Speed-up	Total Speed-up
4	4	3.65	1.28	2.57
	8	3.57	1.60	2.53
	16	3.57	1.85	2.55
	32	3.54	2.10	2.53
	64	3.40	2.22	2.51

The efficiency of the algorithm which is measured as $E_p = \frac{S_p}{P}$ is calculated theoretically as shown in Table 3.2.

TABLE 3.2 The theoretical results of the algorithm

No. of Processors	No. of Paths	Total Speed-up	Efficiency (E_p)
2	4	1.82	0.91
	8	1.82	
	32	1.82	
	64	1.82	
3	8	2.41	0.80
	32	2.41	
	64	2.41	
4	8	2.77	0.69
	32	2.77	
	64	2.77	

We notice that the theoretical and experimental total speed-up are approximately equivalent. The efficiency decreases as the number of processors increases and this is due to the reduction in the usage of the processors in each step of the 2-way merge algorithm.

A performance analysis of this method is predicted together with its performance measurements when run on the NEPTUNE system (Yousif [8]). Thus, we measure the static and dynamic losses of the parallel paths control and the shared data. However, in order to measure the static losses of the parallel paths control, we have to know the number of accesses made by the program to a path per the total number of operations performed in the path. Similarly, with the static loss of the shared data. On the other hand, the dynamic losses of the parallel paths control are obtained directly from the results of the NEPTUNE system and we include here the results of 4 processors when run in parallel. These measurements are the cost of the waiting cycles the processors have spent because no parallel path is available to be carried out by that processor. These calculations are listed in Table 3.3

180 Parallel algorithm design

The parallel path access rate in the 2-way merge algorithm is measured as follows:

Since the number of comparisons is changed in each step of the algorithm, an average of the complexities is taken to represent the total complexity per path. Hence, as the algorithm requires M steps, then the average complexity per path is given by $(2N/M - 1)\log M$. This means that there is one access to a path per $(2N/M - 1)\log M$.

The losses values in Table 3.3 are obtained by using the results of Table 3.4 that includes the timing results of the parallel path accesses on the NEPTUNE system.

If we compare the static parallel path losses obtained from the NEPTUNE system with that of our prediction, we notice that they are in good agreement.

TABLE 3.3. The performance measurements of the 2-way merge algorithm

No. of Paths (M)	Shared data		Parallel path		Parallel path loss	
	access rate	loss	access rate	loss	static	contention
4	2:1 flop	0.2%	$1:((\frac{2N}{M} - 1) \log M)$ flops	0.11%	0.06%	2.6%
8				0.14%	0.11%	2.26%
16				0.22%	0.19%	2%
32				0.36%	0.38%	1.79%
64				0.61%	0.64%	1.58%

However, the contention in Table 3.3 cannot be predicted and can only be obtained from the results of the 4 processors' performance. Also, we notice from Table 3.3 that the parallel path losses increase as the number of paths increases but the contention increases with the increase of the number of paths.

TABLE 3.4. The resources time on the NEPTUNE system

Resource	Time in microseconds
Floating point operation (flop)	~700
Integer	~200
Local memory access	~0.6
Shared memory access	~0.7(average)
Mutual exclusion mechanism	~400
Mutual exclusion blocked	~200
Parallel path mechanism	~800
Parallel path blocked	

11.4 CONVERSION OF IMPLICIT METHODS TO EXPLICIT FORM

Another technique of achieving parallelism in a numerical algorithm is by the use of explicit methods.

However these methods are the oldest methods and suffer from poor stability and convergence characteristics that require unacceptable computer solution times.

The newer implicit methods are better but often we are not able to exploit to the full the implicit parallelism in the solution algorithm.

Hence we must find new explicit methods with improved stability and convergence characteristics.

Consider the simple heat-conduction problem, (Fig.4.1),

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x \leq 1, \quad t > 0, \quad (4.1)$$

with initial conditions, $u(x,0) = f(x), \quad 0 \leq x \leq 1,$
 and boundary conditions, $u(0,t) = g_0(t), \quad 0 < t \leq T,$
 $u(1,t) = g_1(t), \quad 0 < t \leq T.$

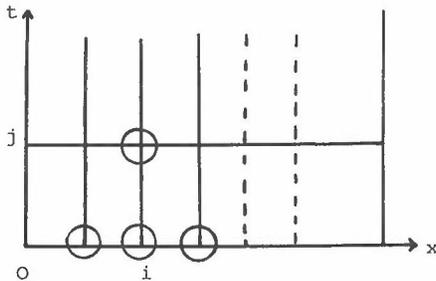


Fig. 4.1

The simplest explicit method uses a forward difference operator approximation to $\frac{\partial u}{\partial t}$ and a central difference operator approximation to $\frac{\partial^2 u}{\partial x^2}$. The formula,

$$u_{i,j+1} = r u_{i-1,j} + (1-2r) u_{i,j} + r u_{i+1,j} + O(\Delta t + \Delta x^2) \quad (5.2)$$

is well known (Fig.4.2) but is unstable for values of $r = \frac{\Delta t}{\Delta x^2} > \frac{1}{2}$.

Hence, the algorithm is ideal for parallel application since every point on the grid can be evaluated at the same time. The method requires long solution times due to the small time step of integration.

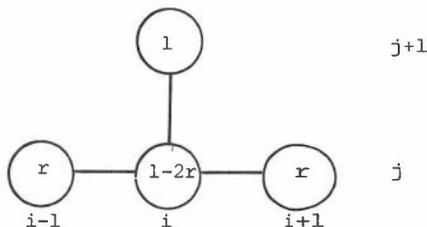


Fig. 4.2

182 Parallel algorithm design

An implicit method uses a backward difference operator approximation to $\partial u/\partial t$ and a central difference operator approximation to $\partial^2 u/\partial x^2$. The equation,

$$-ru_{i-1,j+1} + (1+2r)u_{i,j+1} - ru_{i+1,j+1} \approx u_{i,j}, \quad (5.3)$$

is also well known and is stable for all values of r (Fig. 4.3). However, the algorithm requires the solution of a system of 3 term finite difference equations at every time step in which we are not able to exploit the parallelism to the full.

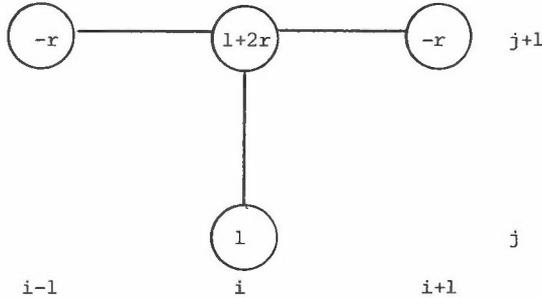


Fig. 4.3

In order to facilitate the solution of these implicit equations, asymmetric techniques due to Saul'yev [9] have been used, i.e. the computational molecule Fig. 4.4 representing the equation,

$$-ru_{i-1,j+1} + (1+r)u_{i,j+1} = (1-r)u_{i,j} + ru_{i+1,j} + O(\Delta t + \Delta x^2 + \frac{\Delta t}{\Delta x}), \quad (5.4)$$

is explicit if solved from left \rightarrow right and the computational molecule Fig. 4.5 representing the equation,

$$-ru_{i+1,j+1} + (1+r)u_{i,j+1} = (1-r)u_{i,j} + ru_{i-1,j} + O(\Delta t + \Delta x^2 - \frac{\Delta t}{\Delta x}) \quad (4.5)$$

is explicit if solved from right \rightarrow left.

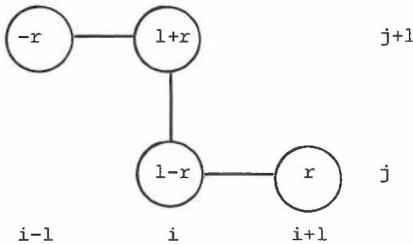


Fig. 4.4

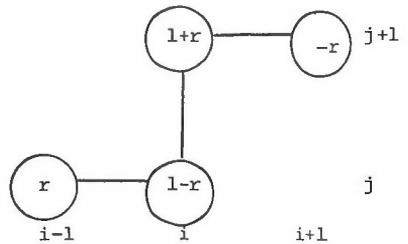


Fig. 4.5

These two schemes are often referred to as semi-explicit formulae.

11.4.1 A New Group Explicit Method

If we now couple the use of the asymmetric equations (4.4) and (4.5) at 2 adjacent points, i.e.,

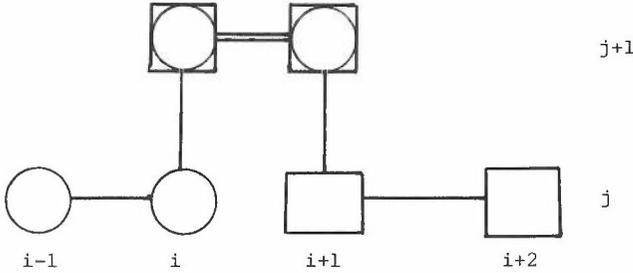


Fig. 4.6

then they result in a (2x2) set of implicit difference equations.

For the group of two points, i.e. $\{i\Delta x, (j+\frac{1}{2})\Delta t\}$ and $\{(i+1)\Delta x, (j+\frac{1}{2})\Delta t\}$ in which equations (4.5) and (4.4) are used simultaneously to calculate the values of u at these points respectively. Therefore, at point $\{i\Delta x, (j+\frac{1}{2})\Delta t\}$ the solution is approximated by

$$-ru_{i+1,j+1} + (1+r)u_{i,j+1} \approx ru_{i-1,j} + (1-r)u_{i,j}, \quad (4.4a)$$

whilst at point $\{(i+1)\Delta x, (j+\frac{1}{2})\Delta t\}$, the solution is approximated by,

$$-ru_{i,j+1} + (1+r)u_{i+1,j+1} \approx (1-r)u_{i+1,j} + ru_{i+2,j}. \quad (4.5a)$$

If we now rewrite equations (4.4) and (4.5) in matrix form,

$$\begin{bmatrix} 1+r & -r \\ -r & 1+r \end{bmatrix} \begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i+1,j} \end{bmatrix} + \begin{bmatrix} ru_{i-1,j} \\ ru_{i+2,j} \end{bmatrix} \quad (4.6)$$

in which the (2x2) matrix of coefficients can easily be inverted so that the equation can be written in explicit form as,

$$\begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \frac{1}{|A|} \begin{bmatrix} 1+r & r \\ r & 1+r \end{bmatrix} \left\{ \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i+1,j} \end{bmatrix} + \begin{bmatrix} ru_{i-1,j} \\ ru_{i+2,j} \end{bmatrix} \right\} \quad (4.7)$$

where $|A|=1+2r$. This simplifies to,

$$\begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \frac{1}{|A|} \begin{bmatrix} r(1+r)u_{i-1,j} + (1-r^2)u_{i,j} + r(1-r)u_{i+1,j} + r^2u_{i+2,j} \\ r^2u_{i-1,j} + r(1-r)u_{i,j} + (1-r^2)u_{i+1,j} + r(1+r)u_{i+2,j} \end{bmatrix}. \quad (4.8)$$

For any ungrouped (single) points near the right and left boundaries equations (4.4) and (4.5) can be used respectively, i.e. for the right boundary,

$$u_{m-1,j+1} = \frac{1}{(1+r)} (ru_{m,j+1} + ru_{m-2,j} + (1-r)u_{m-1,j}), \quad (4.9)$$

and for the left boundary,

$$u_{1,j+1} = \frac{1}{(1+r)} (ru_{0,j+1} + ru_{2,j} + (1-r)u_{1,j}) . \tag{4.10}$$

Finally, equation (4.6) can be easily converted to explicit form resulting in the computational molecule (Fig.4.7).

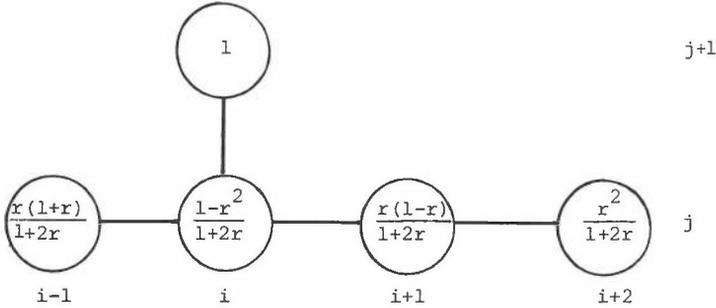


Fig. 4.7

representing the equation,

$$u_{i,j+1} = \frac{1}{(1+2r)} [r(1+r)u_{i-1,j} + (1-r^2)u_{i,j} + r(1+r)u_{i+1,j} + r^2u_{i+2,j}] \tag{4.11}$$

and the molecule,

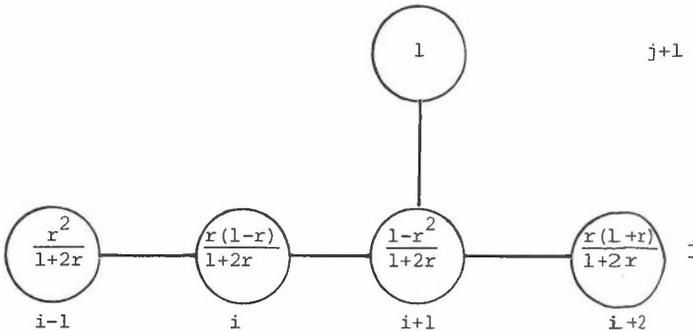


Fig. 4.8

representing,

$$u_{i+1,j+1} = \frac{1}{(1+2r)} [r^2u_{i-1,j} + r(1-r)u_{i,j} + (1-r^2)u_{i+1,j} + r(1+r)u_{i+2,j}] , \tag{4.12}$$

which when used in the alternating group explicit (AGE) method results in a stable explicit algorithm which is ideal for parallel application (Evans & Abdullah, [10]).

Preliminary results indicate that the new alternating group explicit (AGE) algorithm (4.11) and (4.12) not only possesses superior stability characteristics over the standard explicit method but has improved speed-up and efficiency characteristics when programmed for the given problem (5.1) and run on the Neptune parallel MIMD system. The extension of this method to time dependent multidimensional problems is given in Evans [11].

186 *Parallel algorithm design*

7. Knuth, D.E., 1973, 'The Art of Computer Programming: Vol. 3, Sorting and Searching', Addison Wesley Pub.Co.
8. Yousif, N.Y., 1983, 'Parallel Algorithms for Asynchronous Multi-processors', Ph.D. Thesis, L.U.T.
9. Saul'yev, V.K., 1964, 'Integration of Equations of Parabolic Type by the Method of Nets', Macmillan, New York.
10. Evans, D.J., and Abdullah, A.R.B., 1983, 'A New Explicit Method for the Diffusion Equation', pp.330-347, in Numerical Methods in Thermal Problems III, edit. Lewis, R.W. et al, Pineridge Press.
11. Evans, D.J., 1984, 'New Parallel Algorithms for Partial Differential Equations', pp.3-56, in Parallel Computing 83, edit. Feilmeier, M., Joubert, G.R., and Schendel, U., Elsevier Pub.
12. Evans, D.J., 1983, 'New Parallel Algorithms in Linear Algebra', pp. 61-69, in Calcul vectoriel et Parallèle, edit. Bossavit, A., Bulletin de la Direction des Studies et Recherches. Electricite de France.

Chapter 12

Design study for active memory arrays

J. K. Liffe

12.1 DESIGN AIMS

The work described here is intended to offer an attractive range of options in the design of high-performance, high-integrity machines. It is meant to be general-purpose, but the most promising application areas are in support of portable operating systems and languages, high precision software engineering, and problems that can be expressed effectively in SIMD terms.

At first sight the connection with Distributed Systems is tenuous. On closer inspection it will be seen that distribution of function in a generalised sense is proposed only as a last resort, though sufficient control is retained to facilitate such a development. If the object of computer engineering is to take data from the users' files, to transform them, and then to return them for later use or display, movement from file to arithmetic unit and back again is the primary goal and memory bandwidth remains the primary limitation on performance. Having sufficient bandwidth to shuffle the data from side to side in search of processing power is a freak situation encouraged by the introduction of low-cost, low-performance cpu's into the designer's toolkit.

Leaving aside the practical need to handle geographically dispersed files, and assuming that the problem data is in the same box, more or less, as the processor, the problem arises of how to deal with it effectively. If "the problem" is in fact a workload of separate jobs the option of routing them to separate computers is available and is taken on many commercial systems. As the amount of interaction within the workstream increases a solution based on simple partitioning becomes unattractive, and the designer looks to the program memory as the principal weapon in attempting to configure the system to match the workload. Increasing theoretical bandwidth is relatively easy: the aim is to minimise demand. Once that is done the option of increasing throughput by parallelism is still available. The converse approach, hoping that massive parallelism will overcome inefficiencies of the underlying engines or their interconnections, appeals mainly to semiconductor manufacturers.

The traditional roles of random access memory are:
 (a) to extend the function set of the processor by means of stored program; (b) to retain control, structural, and numeric data pertinent to a set of partially executed tasks; (c) to provide the buffer space necessitated by attachment of asynchronous external devices.

In modern systems the development of these roles exerts conflicting demands on memory subsystem design. For example, (c) is eminently suited to distributed, modular design with "message" interconnection, (a) is better suited to shared memory, and (b) hovers between the two extremes. Faced with such variability, the appropriate design strategy is to evolve an engineering model in which all three roles are recognised. Ideally, commitment to one or other mode of operation should be dynamic; at worst it should be deferred until the system is configured.

It has to be admitted that in practice the problem is more often reconfigured to fit the system than vice versa, as illustrated by "array processing" and "reduction" engines. Yet it is frequently overlooked that even in apparently highly specialised machines a substantial part of the workload still requires conventional processing capability: the ICL DAP is backed up by a scalar controller and the ICL 2980 mainframe, the Manchester dataflow computer is supported by a VAX 11/780, and so on. The question arises as to whether one can capture the essential elements of the "novel" architectures within the framework of conventional design. If so, an attractive range of options can be developed by varying the emphasis on one type of function or another without changing the architectural specification.

In the case of array processing that objective has been achieved by the *Active Memory Array* concept to be described here. The user sees a machine with built-in parallel function in the style of the DAP. The engineer can vary the extent to which hardware is committed to support it. Examples will be given of the type of measurement that can be made to guide such decisions.

The Fifth Generation architects, however, have apparently rejected the idea of grafting logic programming onto von Neumann architecture. It is difficult to come to grips with specific requirements because in this field there is as yet no causal chain leading from social demand to the "solutions" which are being offered - one that can comparatively easily be supplied, for example, in passing from energy-saving aerofoil design to CRAY 1. It would be misleading to read too much into current efforts in dataflow or applicative architecture, or even into single-assignment languages and first-order predicate calculus. The "Klips" rate seems an even more dubious parameter than Mips, at least until substantial applications have passed the sampling stage. However, it might be suggested that the computational models that have come into vogue in recent years can be underpinned by quite mundane functions, namely the ability to: (a) scan lists rapidly, evaluating elementary logical and arithmetic functions of their elements; (b) create and destroy lists

freely without incurring heavy penalties in accessing their elements or in collecting garbage; and (c) localise calculations to the extent that they can safely be executed in parallel. If that is anywhere near the mark it is surely unnecessary to throw away the von Neumann baby with the bathwater.

The Active Memory Array (AMA) is allied to an engineering model referred to in Iliffe (1) as the *Pointer-Number Machine*. In the PN machine program structure is recognised at hardware level to an extent that appears to satisfy (a) - (c): not merely in microprogram but at logic gate level. That is to say, (a) and (b) are supported by the parallel function of the memory, and (c) by strict control of task environments. The result can be visualised (Fig 1) as a main memory with an N-word data path, to which a number of vector registers are interfaced. These registers serve as instruction and data buffers for the (scalar) PN processor; they and the processor can be replicated to achieve higher performance until the memory bus reaches saturation. In addition, the vector registers can be presented in parallel to a planar arithmetic-logic and routing network. Placing the parallel component as a shared resource reflects a preliminary feeling for its importance. The relation of this model to others is outlined in Section 12.2.

It was intended ((1), Chapter 13), that part of the vector registers would act as look-ahead cache memory, but the design complexity, and uncertainty as to performance benefit, have delayed progress towards that goal. In the present model the "vector register" is simply the set of working registers addressed in the user's program. This has some interesting consequences in program design which are explained in Section 12.3. The main parameters affecting performance are I_C , the PN machine instruction cycle time; M_C , the memory cycle time; K , the proportion of PN instructions making reference to memory or to the planar ALU; and N , the width in words of the memory data path. In Section 12.4 it will be shown how they affect overall performance.

12.2 RELATED WORK

The novelty of the PN Machine lies in a synthesis of several established techniques which seem at first sight to be in conflict: for example "high integrity" is often associated with performance or cost penalty, and "array processing" with inflexible control or data structures. It will be shown in the next Section how harmony is restored. The underlying techniques are discussed at length in (1), and summarised briefly here.

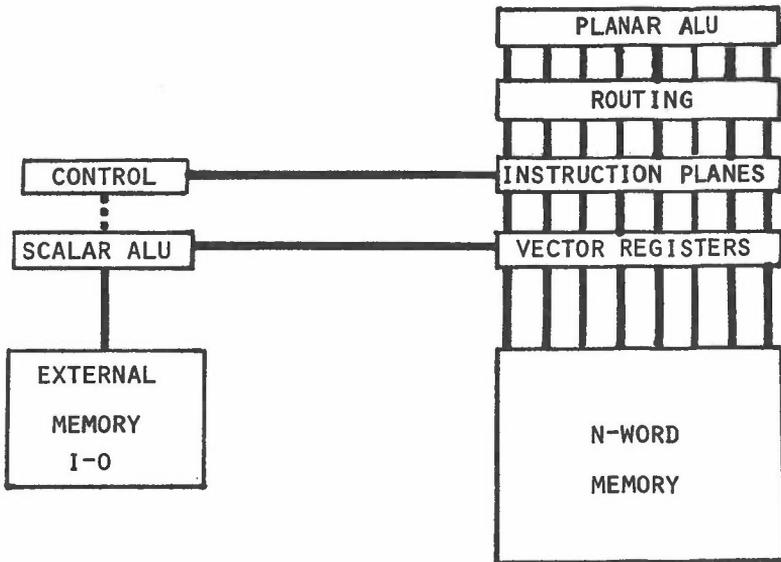


Fig.12.1 Active Memory Array with a single PN Machine.

12.2.1 High integrity design

The aims of high integrity design are to improve system reliability and to reduce software development and maintenance costs. The basic mechanism invoked is a stored element distinct from instructions and data which serves as a pathway to a precisely limited region of program space. This is the "Pointer" component of the PN machine. An array of pointers, possibly mixed with numeric values, delimits the program space that can be accessed by a task at any instant. In the PN machine pointers include capabilities, i.e. identifiers of abstract objects, and physical addresses of data sequences.

Any access control mechanism has to satisfy certain constraints to be acceptable, namely, it must be (a) fool-proof, (b) cheap and fast, (c) flexible, and (d) easy to use. Taking all these constraints into account, a tagged memory and register structure comes closest to meeting requirements. The interpretation of (b) has always been relative to current technology. The earliest machines of this type were constructed using microcoding techniques and compared favourably with their contemporaries. In a sense, however, they were fortunate in having such a large umbrella under which to shelter. The introduction of writable control memories in the early 1970's started a line of research (now seen in the RISC philosophy, Patterson and Sequin (2))

which shifted interest from high level architecture to microsystems. Here the range of options qualifying as cheap and fast is very much reduced.

It will be recalled that access control incurs two sorts of "overhead": that of ensuring that program space is only accessed via the list of current environmental pointers, and that of managing change in the list itself. Indirect addressing of memory via segment tables or resource lists is impractical, which is the reason for placing physical addresses in pointers and mapping the access list into program registers. The points at which the access list changes - typically on procedure calls or at the interface between major software subsystems - require a wholesale adjustment of register contents to reflect the change of environment. In the PN machine this is accomplished very economically with the help of the N-word data path of the memory array.

12.2.2 Microcoding

Achieving controlled memory access within the micro-machine cycle proved to be difficult, but finally removed the major obstacle to the provision of flexible microsystems. The advantages of using language-oriented instruction sets have been demonstrated with varying degrees of conviction. Apart from that there is a growing body of system and application software which is easily transported across machines which can offer an efficient means of imitating a common target language. Experience of such systems seems to suggest that the most effective design strategy is to provide the *choice* of targetting onto microcode, to DEL code, to a fixed instruction set, or to a mixture of all three. Regarding the PN machine instructions as a form of vertical microcode, the above objective has been achieved in the present design. The instruction buffer (Fig.12.1) acts as a microinstruction cache: it will be shown later how it is exploited in the design of interpretive code.

12.2.3 Parallel arithmetic

The principle of the ICL Distributed Array Processor and similar machines has been to avoid the memory bottleneck by placing arithmetic circuits in, or adjacent to, memory itself. The distinctive feature of the DAP is its substitution for a memory module in the mainframe processor in order to serve as a "passive" memory and to escape the overhead of loading and unloading an "attached" device. The functions of the DAP controller enable it to execute parallel algorithms independently of the main cpu but to act as a "slave" to a task in the mainframe: it is not possible, for example, to call a (scalar) Fortran subroutine from within a (parallel) DAP-Fortran subroutine. Although the DAP has been more successful than most array processors in presenting a usable interface to programmers

the rigidity of control and storage structures has been a severe limitation on its applicability.

In AMA design the need for a host is dispensed with completely. The PN machine acts as a controller and carries out all operating system functions including command stream interpretation and compilation. Parallel operations can be freely mixed with scalar, and parallel tasks can be freely scheduled in the interactive workload. In system programming the rigid restriction to an N-word memory plane cannot be avoided, but in application programming generalised array dimensions are supported.

It remains to be seen what level of parallel function can be economically justified, and how practical it is to share parallelism between PN machines. A possible compromise might be to provide non-shared routing and elementary logic, and a shared array of N floating point devices. The present design study is intended to shed some light on such issues.

12.3 PN SYSTEM

The PN machine is designed to support at hardware level the ideas of (a) abstraction, (b) memory management, (c) array manipulation, and (d) interpretation of language-oriented target codes as an alternative to compilation. It follows that a unique combination of features has to find expression in the defining language, while keeping close to convention in other respects. The system language is called "P". By using P in practical applications it is possible to derive quantitative measures of the support required from system functions and hardware, and to make realistic performance estimates.

The following subsections dwell on unusual features of P. For definiteness the description is centred on an implementation using 16-bit words, 32-bit long words, and 36-bit registers. The encoding of P reserves 8 registers for system use, leaving 8 for the application code. A memory plane contains N long words, N registers, or 2N instructions, where typically N=8. The memory depth is 64K planes. Only a single PN machine and single AMA are considered. Input and output devices are addressed as "external memory" of the system: they may be viewed either as individual status or data words, or as more substantial stores such as TV frame or disk buffers. In that way one of the major functions of the memory is detached from program space and associated with the appropriate I-O devices.

12.3.1 Abstraction

Tag coding distinguishes numbers from pointers, and further subdivides the latter into capabilities, addresses, and control pointers. Interpretation of tags is by hardware to ensure integrity of data. Interpretation of capabilities is by program. A user can request a capability for a new class of objects, receiving a *capability-forming-capability* (CFC) in return; a capability for a particular object within

a class is formed by presenting the CFC together with object identification to the system. Misuse is prevented by restricting the circulation of CFC's, which is the responsibility of the users. Forgery is prevented by restricting the use of the tagsetting instruction, which is a privilege of the system.

Capabilities can persist from system start-up until switch-off. Long-term storage in files is not supported. Deletion of objects, including entire classes, is assisted by system functions but generally requires cooperation of the capability class manager to permit early deletion of the representation.

As noted earlier, a set of tagged elements, which might include capabilities, serves to define the execution environment of a task. Certain capabilities are reserved for system use to represent files, storage segments, abstract data types (the CFC's mentioned above), error conditions, etc. A second set of tagged elements known as the *base* represents the *authorisation* environment. A task is executed with reference to a base. A control module is authorised to access only certain named elements of the base, which are declared in the source text. Association of these names with base elements is dynamic, enabling binding to be deferred until execution. It is this mechanism which ensures safe expansion of the execution environment in moving from one control module to another: a program simply requests access to the base element by name, and places the address that is returned in the execution environment. The associative search is carried out by parallel functions.

A stack is associated with each task, organised as a sequence of memory planes. Under program control any subset of the eight user registers or eight system registers can be pushed into memory or, conversely, popped in one memory cycle. A function is provided to "clear" selected registers to the Null value in parallel. A procedure call effectively seals the stack and prevents the called program from unwinding it to get at protected data. In that way the user is given precise control over which elements of the execution environment are passed back and forth between procedures. It can thus be seen that the user benefits from the presence of the AMA even though not explicitly making use of parallel function.

12.3.2 Memory management

Addresses are distinguished by tag as referring to word, plane, or mixed (tagged) sequences. The maximum length of any type of sequence is 4096 elements. The tag further assigns read-write or read-only permission.

Sequences of appropriate tag and length are supplied on demand by the system store manager. A sequence is assigned to a task, is unshared, and is recovered by the system when no addresses refer to it, unless it is *abstracted*, i.e. assigned to the class of objects consisting of storage segments. As with other abstract objects a capability is then created and may be used as the means of controlling

shared access to the segment it identifies. Typically, the segment capability is placed in the base at position p and access is requested by authorised users:

$$x = \text{Access}(p, m)$$

seeks access to segment p , mode m , (update, read-only, etc), and places the result (an address or failure code) in x . On completion,

$$\text{Release}(p)$$

relinquishes use by the current task. Any dangling references (such as x) are annulled by scanning the tagged space of the current task: it is possible to control the propagation of addresses across task boundaries.

One of the more difficult problems caused by using physical addresses in pointers is that of detachment, i.e. of moving the sequence of elements to which a pointer refers out of memory, perhaps to secondary storage. The action of scanning tagged space replacing addresses by suitable markers is not materially helped by parallel operation. It is, however, a good candidate for "microcoding" and that is how it is programmed on the PN machine. It remains to be seen how much machine time it absorbs.

12.3.3 Array manipulation

The data values of parallel arithmetic can conveniently be represented by plane sequences. They can be created dynamically with the help of the system store manager. In vertical mode k planes represent $32N$ numbers of k -bit precision. In horizontal mode the same k planes would represent a vector of kN single-precision (32 bit) numbers.

There are three types of array function in the system language, of which the first two are strongly dependent on array geometry: (a) broadcast operations involving transmission of scalar values to or from store planes; (b) planar routing, which is essentially a two-dimensional shift function; and (c) element-by-element arithmetic and logic. As all higher level functions are built out of these elementary steps it is important to be able to string them together easily, which is achieved in P with the full power of protection, abstraction and other system facilities.

In moving away from geometric limitations it is not difficult to generalise the instruction set and even to arrange that algorithms written and tested on the "real" array should run on interpreted "virtual" arrays. Such an approach is likely to lead to quite significant performance loss, as can be seen by examining the data movement involved in copying one array to another via a (virtual) planar ALU.

The method favoured in the PN system is to develop a library of functions to replace (a) and (b), which can be optimally coded for the real array in the system language. Work on the DAP by Flanders (3) and others gives a strong indication of the type of function that is required.

12.3.4 Instruction sequencing

Use of the planar instruction buffer alleviates one of the main drawbacks of RISC design, namely the difficulty of presenting instructions at the rate required by the ALU. Fig.12.2 illustrates the effect of varying N on the demand for I-plane fetches. It is known, of course, that for

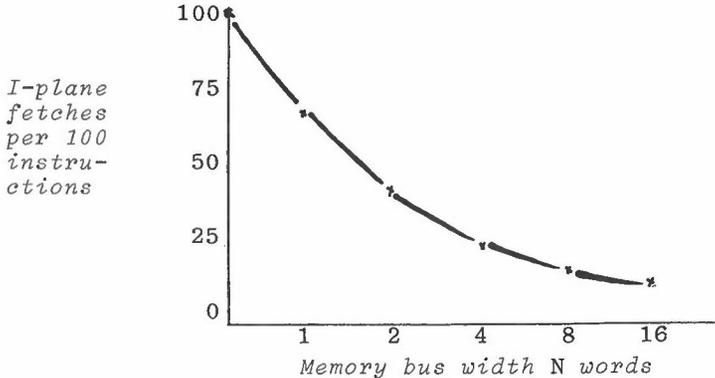


Fig.12.2 I-plane 'miss' rate.

sufficiently large N very high hit rates can be achieved, but here again the logical complexity of large cache memories adds to the difficulty of matching the micromachine instruction rate. A proportion of misses can be covered by conventional pre-fetch techniques; the unavoidable ones are out-of-plane branches, but the margin they leave for improvement is small.

The I-plane structure encourages a view of sequencing in terms of "superinstructions" of $2N$ words. Immediate benefit is gained by allowing P programmers to align code on plane boundaries in order to guarantee uninterrupted execution of short loops. It is also possible to "execute" plane without changing the program counter. For example, interrupts are dealt with not by a vectored branch but by loading the superinstruction of the highest priority interrupt into the I-plane. In effect, the function set of the machine is extended without suffering the overhead of procedure call and return.

A possible application, which has yet to be demonstrated, is in writing interpretive code using multiway branches to I-planes. It will result in a strange hybrid, using the host sequencer and using "soft" function interpretation only when necessary. It does appear, however, to avoid the most costly burden of interpreters, which is payment of sequencing and decoding overhead for even the most trivial operation.

12.4 PERFORMANCE

The AMA-PN-machine combination has been simulated for some years, first on a PDP-11 host but currently on a stand-alone MC68000 system. In order to accommodate substantial applications work is in hand on a hardware-assisted version centred on the AM29116 micromachine (Fig.12.3). This is referred to as "microPN". It does not contain the planar registers or ALU, so all parallel functions are serialised, which represents a performance loss of about $3N:1$ memory cycles on planar operations, and $s:1$ on stack operations, where s is the average number of registers stacked at a time. Program registers are stored partly in the AM29116 and partly in external static RAM. Long operands have to be folded into the 16-bit ALU of the micromachine. Nevertheless, microPN will execute machine functions at about 2.5 Mips compared with 30 Kips on the Mc68000.

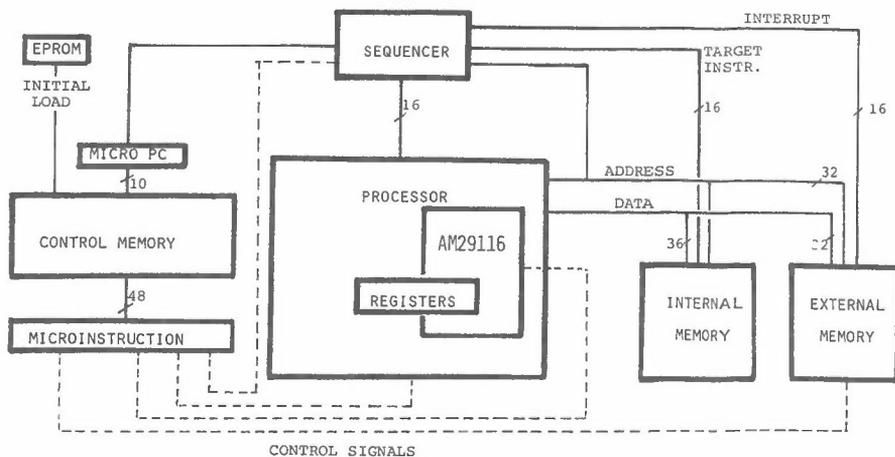


Fig.12.3 MicroPN machine schematic diagram

Crude performance characteristics can be inferred from the data paths of the machine. The most important feature of the design is its ability to switch from one mode of use to another with minimal overhead, for example in moving from serial to parallel operation, from native code to DEL instructions, from sequential to deductive mode of control, from one protection domain to another, or simply from one task to another. Any of these changes can be completed in a half-dozen microinstructions, where it is not unknown for one to take several hundred. Such gross differences are difficult to observe in practice because designers can see them a mile off and attempt to steer round them, e.g. by inventing distributed systems.

Measurements of activity are made by the interpreter. These are incomplete, but show interesting variations in moving from one mode of use to another. Table 12.1 gives results obtained from three different programs: the first is a simple edit-compile-execute session; the second involves data input, permutation, parallel arithmetic, and output; and the third models an intensive parallel algorithm which smooths an image by averaging over sets of four near-neighbour points. I-plane fetches are assumed to be non-overlapped. Stack operations are assumed to be serialised and would be reduced by the factor "s" mentioned above in parallel implementation.

TABLE 1. Measurement of PN machine activity

PN Machine Operations	Problem		
	Edit/compile/go	Array	Grid
Instructions obeyed	100	100	100
I-plane fetch (N=8)	19.4	14.8	10.4
Non-memory instructions	64.3	74.9	62.8
Memory operations:	35.7	25.1	37.2
Scalar Read-Write:	11.9	3.7	1.2
GOTO	3.8	1.9	0.3
Stack (serial)	20.0	12.2	3.2
PE operation	.01	7.3	32.5
PE routing distance:	0	4.1	20.2
Total memory activity (K)	55.1	39.9	47.6

GOTO indicates a change of control module, which involves reference to a segment table in memory. PE routing distance is the total shift of the planar accumulator, North, South, East and West.

Performance estimates are normalised with respect to machine clock cycles. In microPN non-memory instructions require $I_C=2.5$, and memory references give $M_C=4$ (the machine clock is 100nsec). In current PN design $I_C=1.5$ and $M_C=5$ or 3, depending on the speed of memory element used (the machine cycle is assumed to be 60 nsec). Table 12.2 shows the result of evaluating

$$\begin{aligned}
 C &= I_C * (\text{no of non-memory instructions}) \\
 &+ M_C * K \\
 &+ \text{PE routing distance}
 \end{aligned}$$

which assumes that planar shifts take one machine cycle per step.

TABLE 2. Estimation of PN machine performance

(N=8)	Total machine cycles(c) per 100 instructions		
	Edit/compile/go	Array	Grid
$I_C=2.5, M_C=4$	381	351	368
$I_C=1.5, M_C=5$	371	316	352
$I_C=1.5, M_C=3$	262	236	257
$I_C=2.5, M_C=4$, serial	382	1227	4433

The last row of Table 12.2 shows the result (in microPN) of dispensing with the parallel ALU and planar registers. Current work is aimed at improving the accuracy of the simulation relative to possible implementations and widening the range of workloads to which the system can be applied.

12.5 ACKNOWLEDGEMENTS

The work described here has been partially supported by the Science and Engineering Research Council under Grant GR/C/32126. The author would like to acknowledge the contribution of Mr. P. Griffiths to the design of microPN.

12.6 REFERENCES

1. Iliffe, J.K., 1982, 'Advanced Computer Design', Prentice-Hall International, London.
2. Patterson, D.A., and Sequin, C.H., 1982, 'A VLSI RISC', Electronics Research Laboratory Memorandum No. UCB/ERL M82/10, University of California, Berkeley.
3. Flanders, P.M., 1982, 'A Unified Approach to a Class of Data Movements on an Array Processor', IEEE Transactions on Computers, Vol.C-31, No.9, 809-819.

Hardware and software for parallel update of raster graphics images

I. Page

13.1 INTRODUCTION

The user interfaces of complex information processing systems such as those for computer-aided design, computer-based training and automated office systems are rightly receiving an increasing amount of attention from computer scientists. However, many of the new techniques being employed in user interfaces, such as dynamic windows, pop-up menus, dragging and animation require a large amount of computational power to support them properly. In the type of highly interactive information systems mentioned above, this computational power is often provided by a powerful, single-user, graphics-based workstation which also has to run the application programs. Also, the software structures needed to run such a user interface will often become extremely complex in an attempt to cater for the entirely reasonable desire of the user to apply any operation that he knows about, in any context that he happens to be in.

Considerable simplifications in the construction of such powerful user interfaces can be obtained by using a display list as an interface between the applications programs and the screen image. The display list is a high-level data structure stored in the address space of the host computer which represents the screen image in terms of windows, groups of windows, strings of text, bitmap pictures etc. The separate (often concurrent) applications programs can then interact with the data structure very easily. Moving a window, for example, becomes a simple matter of changing the x & y co-ordinates of the window in the display list and changing the z-order of the windows might only involve re-ordering a window list.

This approach considerably simplifies the task of constructing complex interactive systems but it also increases still further the amount of processing power needed to support the screen image. We would now need to continuously refresh the screen image from the data structure at a rate of up to 25 times per second and this clearly needs special-purpose hardware support.

A short investigation showed that virtually all of the computation time is spent in a software procedure often known as 'RasterOp' (Newman (1)). This operates on rectangular sub-areas of bitmaps and can move and combine such bitmaps under various Boolean operations. Despite the fact that it is conceptually very simple, RasterOp is an extraordinarily powerful procedure and can even perform many operations which are not intuitively obvious (such as rotating bitmap pictures and doing fast scalar arithmetic! Guibas (5), Goldberg (7)). We decided to

200 Parallel update of Raster graphic images

support this procedure and also the display list interpretation with special purpose hardware. The hardware system that has resulted from this we fondly know as 'DisArray' (short for **Display Array**).

DisArray uses a two-dimensional interconnected array of very simple computational elements each one paired with its own local memory. The DisArray hardware has the ability to manipulate two-dimensional areas of a bitmap (up to 16 x 16 bits square) in a single memory/processor cycle. DisArray can use this ability to form new screen images in a very short time by rapidly assembling the components of the picture (individual characters, line segments, half-tone shaded areas, whole windows etc.) into a screen-sized bitmap which can then be displayed on a conventional C.R.T. monitor.

This approach has the advantages of high speed since the memory and computational elements are very closely coupled, and of high throughput because of the high degree of parallelism and the two-dimensional interconnections within the array.

13.2 A QUICK INTRODUCTION TO RASTEROP

As noted above RasterOp operates on rectangular arrays of Pixels (Picture Elements). An over-simplified Pascal version of the RasterOp algorithm is shown below. This version ignores the problems caused by having overlapping source and destination rectangles and assumes that pixels are only ever Black or White (false or true). The data type 'raster' is not defined, but is essentially a two-dimensional array of Boolean variables corresponding to a picture. The rasters are only manipulated through the access functions 'GetPixel' and 'SetPixel' which are not defined here but have the obvious meanings. The 'operation' parameter has the following effect on the destination rectangle :

```
Black : Dest := all black
White : Dest := all white
Copy : Dest := same as source rectangle
Invert : Dest := logical inverse of source rectangle
ROR : Dest := logical OR of source and destination rectangles
RXor : Dest := logical EXOR of source and destination rectangles
```

```
procedure RasterOp (operation : integer;
  var Dest : raster; xd, yd, width, height : integer;
  var Source : raster; xs, ys : integer);
const White = false; Black = true;
var x, y : integer;
begin
  for y:=1 to height do
    for x:=1 to width do
      case operation of
        Black : SetPixel (Dest, x, y, Black);
        White : SetPixel (Dest, x, y, White);
        Copy : SetPixel (Dest, x, y, GetPixel (Source, x, y));
        Invert : SetPixel (Dest, x, y, Not GetPixel (Source, x, y));
```

```

ROr      : SetPixel (Dest, x, y,
                GetPixel (Source, x, y) Or GetPixel (Dest, x, y));
RXOR     : SetPixel (Dest, x, y,
                GetPixel (Source, x, y) <> GetPixel (Dest, x, y));
end
end;

```

This would be a hopelessly inefficient implementation of RasterOp, but it does effectively show the nature of the algorithm and more sophisticated versions of it still retain these same essential features. Also, this version has only some of the 'operations' that a real implementation might offer. Further details of the algorithm can be found in Newman (1).

13.3 DISPLAY LIST APPROACH

The display list is an abstraction of the image to be presented on the screen to the user. It acts as an interface between the application programs and the refresh process which has to construct a screen image. The design of the display list is therefore a crucial issue, since it is at the heart of all the display operations and these need to be performed quickly. Also its structure needs to be appropriate both for the different applications programs and for the display manager. This always means that trade-offs have to be made since the requirements of applications programs and the refresh process are significantly different from each other.

A simple example of a display list in which the major structural elements correspond to 'windows' (or 'pages' or 'pieces of paper') on the screen is shown in Fig. 13.1. The whole structure is a list of window descriptors which each store the attributes and the contents of one window. The order of the windows in the display list is the same as the z-ordering of the windows when shown on the screen. Thus the screen image can be generated using the painter's algorithm (ie. back-to-front fill) by traversing the list from beginning to end, filling in the contributions to the screen from each window in turn. Fig. 13.2 shows the screen image which corresponds to the display list in Fig. 13.1.

In this simple example, the windows contain only text and so the data structure for each window includes a list of text strings which comprise the text to be put in the window. An application program operating on one of the windows can now be given a pointer to 'its' window descriptor and can move it on the screen by changing the x, y offset values in the descriptor. It can similarly change the background colour of the window or its size or the font of the text by altering the descriptor appropriately. The text in the window can be scrolled by simply cycling the pointers in the array of text string pointers.

The display processor needs to continuously scan this display list and generate a screen image from it. It can be appreciated that this process involves little more than scanning the display list in order, performing some simple clipping operations (to window and screen boundaries) and

then performing the RasterOp operation on whole windows (to clear them to their background colour) and then to use RasterOp again for each of the characters in the text strings to copy the bitmap images of the characters from a font table into the window area. Thus we need a relatively fast machine to traverse the display list and a super-fast machine to implement the RasterOp commands generated from the display list interpretation.

As a historical note, around 1977 at QMC we built a hardware system known as the QMC Text Terminal which used exactly this style of display list and generated 50 completely new frames per second using a purpose-designed bit-slice processor (Page (8)). It produced a fully-animated, colour display of a desk-top with 'pieces of paper' but was restricted to showing only textual and block graphics information. A large part of the motivation for the DisArray project stemmed from the success of the Text Terminal experiment and our desire to have a system which would perform similarly with multi-fonted text and more complex graphical images.

13.4 DISARRAY, RASTEROP AND LINE DRAWING

13.4.1 RasterOp

Despite its simplicity, RasterOp is an extraordinarily powerful primitive and, when it is well-supported in the hardware in such machines as the Alto (Thacker (2)) and the Perq (Three Rivers (3)), it gives those machines a very good interactive graphics capability. However, we wish to improve significantly on that performance by employing parallelism in the hardware. We would also like to find a form of parallelism which can offer even greater throughput by simply increasing the amount of parallelism in the hardware.

In its simplest form on a conventional machine, the inner loop of RasterOp would take a word-aligned word from store (part of the source) and move it to a bit-aligned word in store (part of the destination). This bit-aligned word will usually fall across the boundary of two store words and thus require two read/modify/write cycles to update it. This effect slows down the algorithm but it can be ameliorated by pipelining the data over a number of such inner loop cycles.

To improve their graphics speed, machines such as the Perq have a micro-programmed RasterOp. Also, they make special arrangements to increase memory bandwidth (by employing wide data highways), and have a barrel shifter to do the alignment to bit boundaries. This is probably about the limit of support that a conventional processor can offer to RasterOp, but there are (always!) good reasons for wanting to increase its speed still further. It should also be noted that such a conventional machine would always be very much better at doing short, fat RasterOps than tall, thin ones. Such an implementation will perform most poorly (in terms of pixels/second updated) when drawing a pixel-wide vertical line. In this case only one bit is being usefully updated

on each memory cycle. Paradoxically, in this particular case, this bandwidth degradation only gets worse as the machine data paths are made wider in an attempt to improve the general throughput of RasterOp.

In fact, in the graphics world, there is no reason why RasterOp rectangles should be of any particular shape and the theoretically most efficient shape for the basic word is thus a square, which is equally optimised for both worst cases of the tall, thin and the short, fat rectangle.

DisArray uses such a square word, known as a plane, to support RasterOp. The analagous inner loop of a DisArray RasterOp, takes a (plane-aligned) plane from memory and moves it to a bit-aligned plane, which will usually fall across four actual memory planes. A series of diagrams (Fig. 13.3) shows a sequence of DisArray operations to perform a single step of the inner loop of a RasterOp operation.

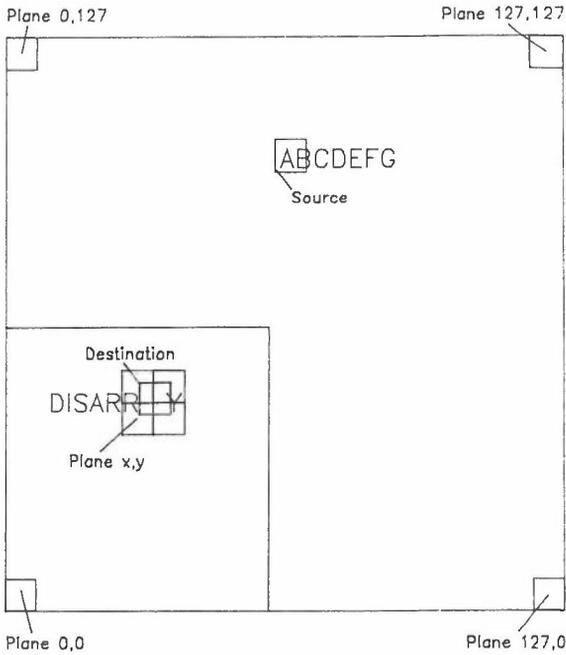
13.4.2 Line Drawing

A first sight it might appear that the DisArray style of processor offers little to a line drawing algorithm, such as the ubiquitous DDA (Newman (1)). However, such is the power of RasterOp that it can be used to great effect in a DDA style algorithm which plots more than one point at a time.

The basic idea is to pre-compute a set of short line 'strokes' at various angles and to keep these in a table. Whenever an arbitrary line is to be drawn, it can be formed by choosing the appropriate stroke(s) from the table and using RasterOp to place them appropriately in the image. In our software, the strokes are pre-computed into DisArray planes and a procedure identical to the inner loop of RasterOp places these strokes into the image being constructed. Thus 16 points along a straight line can be plotted in parallel in exactly the same time as a RasterOp cycle. Further details on this approach can be found in Gupta (9).

13.5 A SIMPLE EXAMPLE

In the example shown in Fig. 13.3, the letter 'A' is to be copied from its current position in a font-table to a position within the portion of DisArray memory which is (currently) being refreshed onto the screen. The user would thus see the word 'DISARRAY' being completed on the screen (Fig. 13.3a). DisArray has a 16 x 16 register, known as the Q-Register. Fig. 13.3b shows the contents of the Q-Register after a single memory cycle which loads the letter 'A' from the appropriate source plane. During this same cycle, the Row and Column masks shown in fig 1b are ANDed bit-wise in each processor to obtain a 16 x 16 bit mask. This mask defines the limit of validity of the source region within this source plane. A single 'read' operation will read the source plane from memory, AND it with the 16 x 16 mask and then store the result in the Q-register.



The whole DisArray address space regarded as a single bitmap. The word 'DISARRAY' is being completed using a particular bitmap-defined font.

Fig. 13.3a

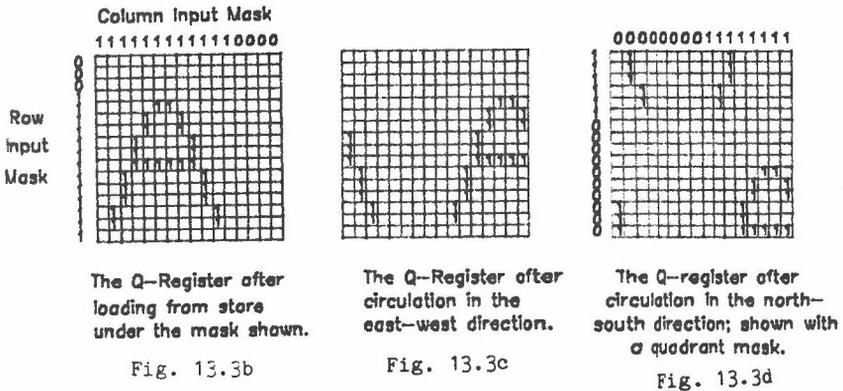


Fig. 13.3 : Steps within a DisArray RasterOp

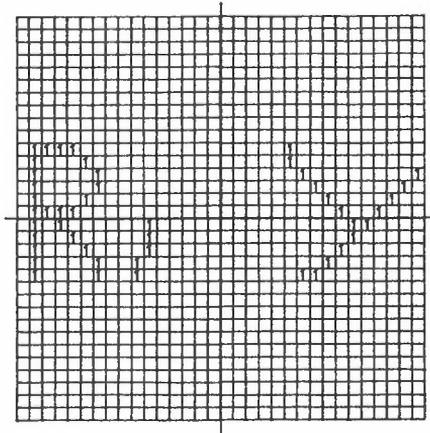


Fig. 13.3e The four destination planes after a read/OR with Q-Reg/write cycle (under the quadrant mask) to plane m.

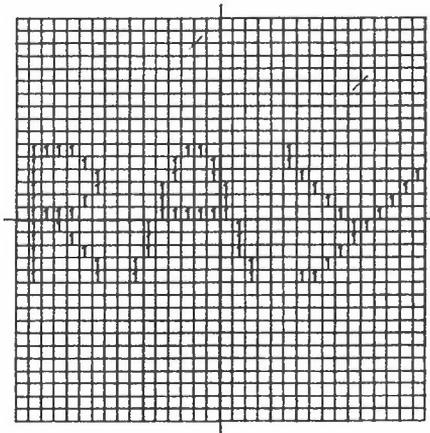


Fig. 13.3f The four destination planes after three more similar cycles with appropriately altered addresses and inverted row/column input lines.

Using the nearest-neighbour connections between the processors, the Q-Register is shifted by appropriate amounts, first in the east-west direction (Fig. 13.3c) and then in the north-south direction (Fig. 13.3d). Fig. 13.3d also shows the state of the Row and Column masks which will be needed for the subsequent operations. These operations are on the four separate 'quadrants' of the letter 'A' which lie in the four corners of the Q-Register. Notice that the logical AND of these two masks selects the bottom left-hand corner of the 'A' (now in the top right-hand corner of the Q-Register). The other quadrants are selected by appropriately inverting the row and/or column masks.

Fig. 13.3e shows the state of the four destination planes after a single read/modify/write cycle involving the lower left of these four planes. The shifting performed above has resulted in the bottom left-hand corner of the 'A' being correctly aligned with the top right-hand corner of the destination plane where it eventually needs to go. A single DisArray cycle reads the previous state of the destination plane and writes it back unchanged, except in the area designated by the quadrant mask, where the contents of the destination plane are replaced by the logical OR (in this case) of the corresponding part of the Q-Register (the source) and the previous contents of the destination plane.

Fig. 13.3f shows the state of the destination planes after a further three read/modify/write steps and the desired operation has been completed. We call this sequence of steps a 'RasterOp Cycle'. With slight variations to cater for edge effects, this RasterOp cycle can be repeated many times to deal with source rectangles which consist of many planes and the example presented is indeed representative of the inner loop of a generalised RasterOp.

13.6 THE ARRAY PROCESSOR

13.6.1 Overview

The DisArray hardware has an array of 16 x 16 Processing Elements, which are each simple 1-bit processors, each having a 16K x 1-bit local store. The array as a whole therefore deals with 256-bit square words, known as planes. All Processing Elements execute the same instruction simultaneously on their local data, making it an S.I.M.D. (Single Instruction stream, Multiple Data stream) machine. The Processing Elements each have connections to their four nearest neighbours in the array so that planes can be shifted bodily in the four orthogonal directions, one bit position at a time. The edge connections are toroidal so that the shifting is in fact circular in the two dimensions. The architecture is similar to the I.C.L. Distributed Array Processor (Reddaway (4)), which partly inspired this project.

The Array Control Unit turns a set of RasterOp parameters into an appropriate sequence of array operations to implement a particular 'call' of RasterOp. An outline diagram of the array system configuration is given in (Fig. 13.4), but with a much reduced size of

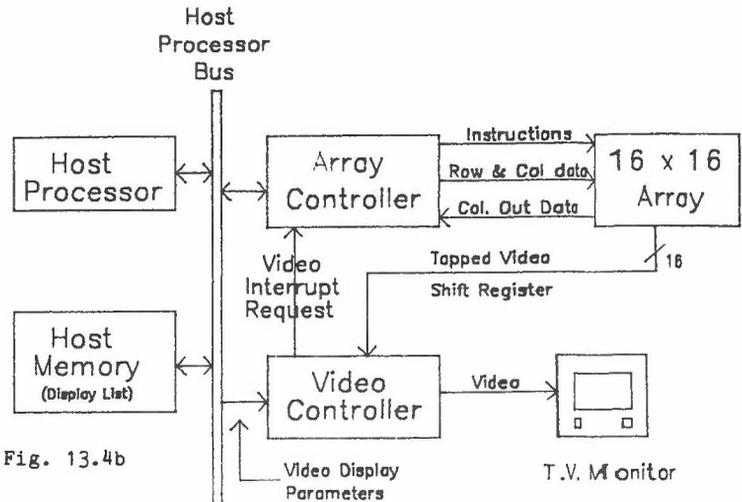
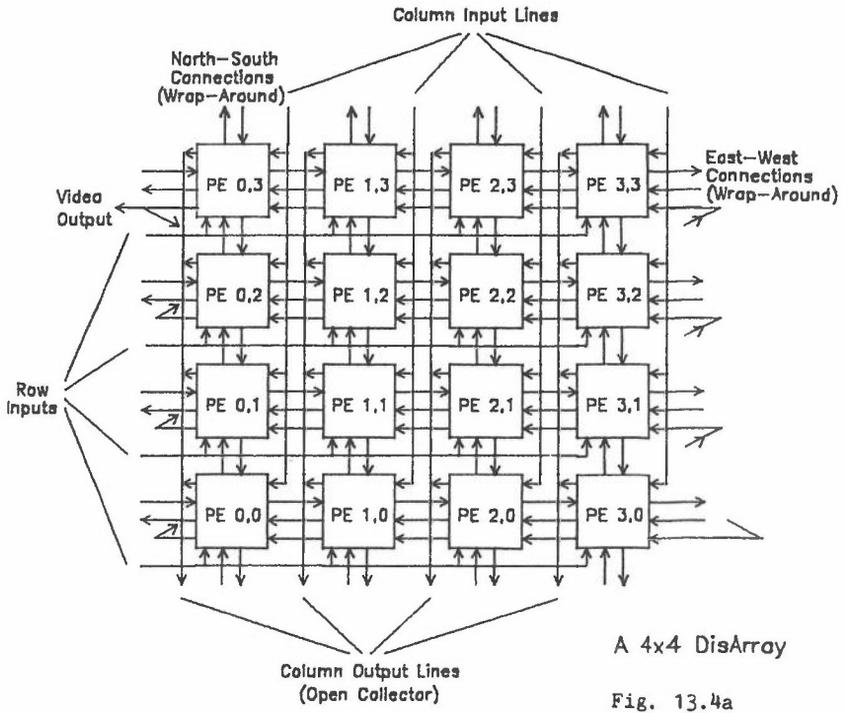


Fig. 13.4 : Architectural Overview of DisArray

array for reasons of clarity. The Array Control Unit is also responsible for autonomously interpreting the display list and turning it into the implied sequence of RasterOp procedure calls.

A host processor runs the application programs which create and manipulate the display list. Currently, the system is in fact running in a stand-alone mode without a properly integrated host processor. We hope to build a DMA link to an Orion (a 32-bit micro-programmable engine running Unix) in the near future.

13.6.2 The Processing Elements

A block diagram of a single Processing Element is shown in Fig. 13.5. The processing is done by the ALU, which is in fact an 8:1 multiplexor. This, in effect generates an arbitrary function of the Q-Register output and the memory output. However, one of two such arbitrary functions is selected on the basis of the logical AND of a row-derived and a column-derived input line. Together, these are normally used to select an arbitrary sub-rectangle of the array based on one or other of the corners of the array. Such sub-rectangles are known as quadrants.

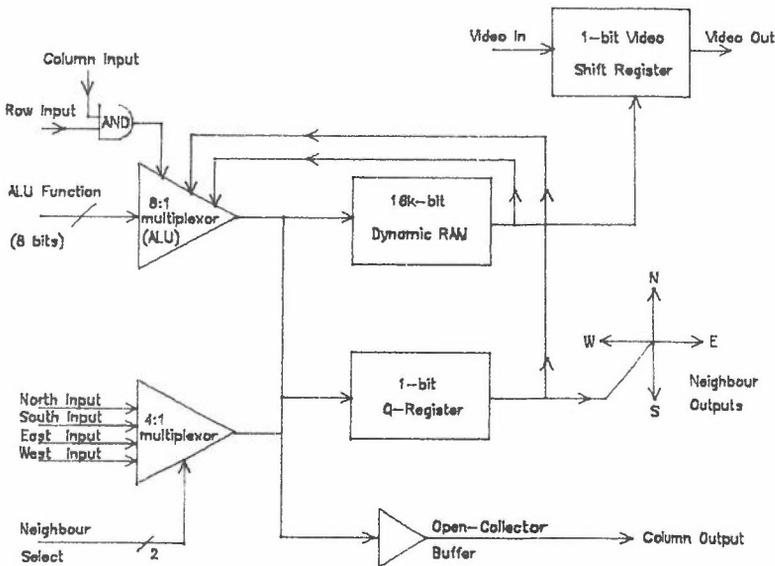


Fig. 13.5 : A DisArray Processing Element

210 Parallel update of Raster graphic images

Two 16-bit strings, each consisting of a single group of consecutive 0's and a single group of consecutive 1's, applied to the row and column input lines are sufficient to define a single quadrant (ie. that area where both Row & Column inputs are 1). The other three similarly-aligned quadrants can be selected by inverting the bits in either one or both of these bit strings.

This arrangement allows some arbitrary function to be performed in the region of the selected quadrant. Often, however, the rest of the processors outside of the quadrant will have to perform some simple (non-useful) operation such as copying whilst this is going on.

There is a single 1-bit register (the Q-register) which holds the result of computations and is also the holding register for array shifting operations. The design has optimised the nearest neighbour shifting by putting only the Q-register and a 4:1 neighbour selection multiplexor into the shift data path. We hope to enhance the system at some future date with one or more additional Q registers.

The local store is a 16 Kbit dynamic RAM whose control inputs (including the address lines) are derived from the Array Control Unit and whose data input is fed from the output of the ALU. A 16-bit output bus from the array is generated from 16 sets of 16 open-collector outputs in each column of the array being wired together. By selecting only one row of the array, using the row and column input lines, a single 16-bit word can be output from the memory to this bus. This is used to map the DisArray memory into the host processor's address space. Similarly, 16-bit data from the host can be written into array memory by putting the data on the column input lines and selecting just one row using the row input lines.

As a bonus, the open-collector bus can be used to support some simple host-accessible content-addressing of memory. For example, arranging 16-bit words column-wise in store and broadcasting a 16-pattern along the row input lines the host can look for a match in any one of the 16 columns simultaneously, the corresponding column output line saying whether a match was found.

The control signals for the Processing Elements are provided by the Array Control Unit and are copied identically to each processor. This set of control signals thus makes up a single array instruction. This instruction consists of :

1. The memory (plane) address to be used.
2. Memory control signals (RAS, CAS etc).
3. The two Boolean functions for the ALU.
4. The nearest neighbour selection.
5. Q-Register latch signal.
6. Multiplexor enable signals.
7. Miscellaneous edge control signals.

¹ In fact, the memory address to each processor is in fact systematically altered under both the address-staggering and the quadrant-addressing schemes outlined later.

13.6.3 The Basic Array Cycle

The most general type of array instruction is one involving a read/modify/write operation on a single plane in memory. Such an instruction performs the following function:

- a) Reads the contents of a 256-bit plane from memory.
- b) Examines the state of the row and column input lines. These usually contain masking information and the logical AND of these inputs is generated to select a quadrant.
- c) Applies an arbitrary Boolean operation between the contents of the plane and the Q-register in the region of the selected quadrant and applies another arbitrary Boolean operation outside that quadrant.
- d) The result of the computation cycle c) is optionally written back to the same location selected in a), and/or latched into the register and/or sent to the column output lines. This read/modify/write cycle is terminated early whenever possible.

13.6.4 The Array Control Unit

The Control Unit has the task of generating the instruction stream which controls the array. This instruction stream is generated either from interpreting the display file or by directly executing RasterOp procedure calls from a process running in the host processor (usually a screen manager process). Since the control unit is micro-coded, both of these models of operation and many others are possible, simply by re-loading the micro-code store.

The control unit consists of an AMD29116 16-bit datapath chip, a 2910-based sequencer and a 4k x 32-bit writable micro-code store. It can execute independently of the array processor but is the sole means of initiating an array processor cycle. It has DMA access to the memory of the host processor and this is used to give the controller read-only access to the display list. All of the array processor registers can be directly written by the controller in order to set up array instructions and edge data. Additionally, there is a 4k x 16 local cache store which can hold working variables and copies of parts of the display list needed during interpretation.

13.6.5 The Refresh Controller

The refresh controller autonomously takes a bitmap from store by stealing array memory cycles. This produces 256-bits of data which is then broadside-loaded into a spirally-arranged video shift register which runs through the whole array. This data is then asynchronously clocked out directly to the monitor at video speed.

212 Parallel update of Raster graphic images

There is will always be a fundamental mismatch between the requirements of RasterOp and those of video refresh; the former needs to operate on square words and the latter needs to operate on very long thin words (scan lines). DisArray has an address staggering scheme which overcomes this mismatch without any need to resort to buffering of the video output data.

DisArray currently supports a 512x512 pixel bitmap display with 4 bits per pixel. A colour map RAM selects the 16 pixel colours from a palette of 4096 colours. This video output format is relatively arbitrary and can be easily changed. There is currently over 4 times this video bandwidth available if necessary, to support either higher-resolution displays or more colours.

13.7 ADDRESS STAGGERING SCHEME

To achieve direct video refresh from the array without the use of output buffering needs some re-organisation of storage. Clearly, considering scan line 0, all of the bits contributing to this line are in row 0 of the array.

To get 16 consecutive 16-bit segments of scan line 0 out of the array simultaneously (which is what the t.v. monitor requires), these segments must necessarily then be in different rows of the array. This can be accommodated at no extra software or hardware cost by arranging that horizontally consecutive planes are stored such that they are circularly shifted respectively southwards by one row. Now, it is only necessary to arrange that each row of processors gets the appropriate refresh address on any video refresh cycle. These addresses can be simply formed from a single video refresh plane address by adding the row number in the array to the refresh address, but not allowing the carry to propagate past the bottom four bits. This is equivalent to fetching a plane of data out of array memory angled at 45 degrees rather than the usual horizontal alignment.

The only other remaining problem is that the 'origin' of the 256-bit scan-line segment that gets loaded into the video shift register is also shifted. However, this is easily overcome, since the shift register is spiral and all we need to do is to arrange for 16 tapping-points on the shift register at the end of each row and to select one of these tapping points with a multiplexor to get the correct video bit stream.

¹ In fact, with the memory mapping arrangement in DisArray the appropriate address to be sent to each row of processors is :

$$(\text{RefreshAddress} - \text{RowNumber} - 1) \bmod 16$$

This is easily provided from a 4-bit alu slice on the low-order 4 bits of the memory address bus on each row of processors.

13.8 QUADRANT ADDRESSING SCHEME

As discussed above, the inner loop of RasterOp entails adjusting a plane to bit boundaries in both directions and then performing a read/modify/write cycle on each of the four quadrants involved. On each of these cycles the array is not performing useful work in the other three quadrants. The four memory cycles can be compressed into a single cycle if the processors in the four quadrants could each receive a different address.

There are a number of possible ways to achieve this, three of which are outlined here :

1. Broadcast all four addresses in turn along the common memory address lines and use strobes generated on a per-quadrant or per-processor basis to cycle the memory chips. This is a simple extension of the method of time-multiplexing of the address lines already in use because of the nature of dynamic ram addressing.
2. Build memory chips¹ with some intelligence in the address paths. What is needed is the ability to store a small number of addresses on the memory chip and to optionally combine one of these with the incoming address in a simple adder with outside control of the carry-in signal. In this way, with a current memory address m , and a stored address s , the chip could access the following locations under the control of two extra control signals :

$$m, m+1, m+s, m+s+1$$

This is precisely the pattern of addresses required by RasterOp.

3. Partition the memory address into two parts; an x-plane-address and a y-plane-address. This institutionalises what the software often does anyway, which is to regard the array memory storage as a single, large two-dimensional bitmap.

This means that it is possible to generate the two x-parts of the addresses and broadcast the appropriate one of them down each of 16 sets of column-wise address busses. If the same is done with the y-parts broadcast row-wise, then the address required by each processor is then formed by concatenation of the x-part and y-part that passes through that processor. This scheme has the considerable benefit of requiring no extra hardware in each processor.

In fact, none of these quadrant addressing schemes have yet been implemented in the hardware. We may however implement the third method in the not-too-distant future although we would really prefer option 2 if we could get access to the technology required to produce the necessary 'smart' memory chips. We have already successfully designed

In fact, if we had the capability of building a reasonable size of memory chip, then we would put somewhat more intelligence than this into the address lines and also put a processing element onto the same chip. This results in a smart memory chip with many other useful applications.

and fabricated out first chip¹ but the fabrication facilities we have available are not nearly good enough for us to create a useful size of intelligent RAM.

13.9 DISARRAY PERFORMANCE

The array can shift the Q-Register in any of the four orthogonal directions at 30MHz, giving a total bit shifting capability of over 7 Gbits/second. This operation is the basic mechanism for aligning a source plane with the (four) destination planes and it is the two-dimensional analogue of the use of a barrel shifter in the data paths of a one-dimensional RasterOp processor.

The array has a rather leisurely 600ns read/modify/write memory cycle time. This is slower than it need be because of the low speed of the dynamic rams chips² that we used. This gives the array a basic computational rate (Memory := Memory Op Register) of just over 400 Mbits/second (plane-aligned). This could easily be tripled using current memory chips to 1.2 Gbits/second.

Timing tests on the prototype have not yet been carried out as we have only just finished the first encoding of the basic microcode software. However, we expect a to achieve speeds something like the following :

	Current Hardware	With Quadrant Addressing	With 200nS Store	With Both
Character-sized RasterOps/sec	200k	300k	300k	500k
Large scale RasterOp > 10 kbits. Rate in Mbits/sec	80	160	200	260

These figures ignore the video refresh overhead, which is variable³ depending on the screen size.

¹ A 32-bit in, 16-bit out barrel shifter with two-level pipeline input registers for aiding RasterOp on conventional 16-bit micros.

² The ram chips were purchased over four years ago, before a rather long break in the project when no manpower was available for construction.

³ For example, with a 1024x768 picture the overhead is about 200k memory cycles/sec.

13.10 DISARRAY2

13.10.1 The Next Generation

It is our intention to build a second generation DisArray machine with certain advanced features. This will take the form of a fifth generation workstation in which a microprogrammed Powerful Personal Computer is completely integrated with a DisArray processor. DisArray2 is currently an outline design for such a system, based on newly-available, very powerful LSI components, which could improve on the performance of the current DisArray by a factor of about 5 and also provide (at peak) about 300 Mips of vector processing on 16-bit quantities. We will introduce the reader to this design in a number of stages.

13.10.2 Surface Shifting

One of the areas in which DisArray can be speeded up is by speeding up the serial shifting that is required to align planes to bit-boundaries. In a conventional machine the solution would be to replace the serial shifter with a barrel shifter.

Happily, a similar solution is possible in the two-dimensional case. We simply replace all of the horizontal neighbour connections in a single row of the array with a barrel shifter. This then gives us a fully-connected arrangement where any processor has a direct connection to any other processor in the same row. We repeat this for every row in the array using a total of 16, 16-bit barrel shifters. We also repeat the pattern along the 16 columns of the array giving each processor immediate access to any other processor in the same column. In fact, there may be no need to have access to the intermediate result between a horizontal shift and a vertical shift. In this case, at each processor location the processor can send data to the appropriate input port of a horizontal barrel shifter. The corresponding output port of that horizontal barrel shifter is directly connected to the appropriate input port of a vertical barrel shifter, whose corresponding output is then fed back to the same processor.

This composite shifter, comprising 32, 16-bit barrel shifters, is a two-dimensional analogue of the one dimensional barrel shifter. I have dubbed this novel structure a surface shifter. A diagram of an example 4x4 surface shifter appears in Fig. 13.6. This is implemented using 8 4-bit barrel shifters connected together in a pipeline type of arrangement with an array of 4x4 Processing Elements.

This arrangement can obviously speed up the shifting¹ but at some cost. In fact one of the nice features of the simple nearest neighbour-connected array is that it can be extended indefinitely. The surface

¹ However, using currently available components this speed-up is not quite as good as one might at first suppose since serial shifting can be made quite fast (DisArray currently runs at 30 MHz).

shifter can probably only reasonably be implemented with at least each barrel shifter being totally implemented on one chip. This means that the maximum size of the array would be limited by the number of pins on a package. Currently, we do not have access to the production facilities that would be needed if we made a 32-bit barrel-shifter/processor chip.

Another advantage emerges if it is possible to set a separate shift constant for each row and column shifter simultaneously. Consider the case in which the row number of each row is used as the shift amount for a circular shift in that row. Considering only the horizontal component of the shift, what we have achieved is a skewed shift of the data, with wrap-around. Row 0 is unchanged, Row 1 is circulated by 1 bit etc. I have called this a skew-circular surface shift. In this case it is a horizontally-based shift, but it obviously has a vertically-based counterpart. By applying the following sequence of shifts to a plane :

1. Horizontal skew-circular surface shift.
2. Vertical skew-circular surface shift.
3. Horizontal skew-circular surface shift.

the effect is to completely rotate the contents of the plane by 90 degrees in only three machine cycles (Guibas (5), Goldberg (7)).

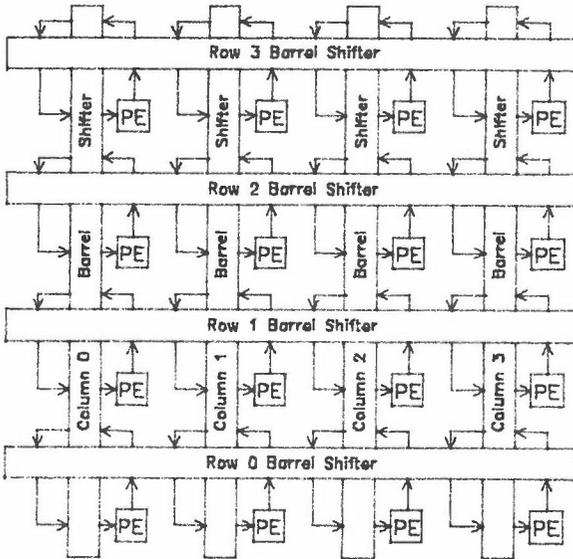


Fig. 13.6 : A 4x4 Surface Shifter

13.10.3 Processor Element Implementation

Having decided to use the surface shifter concept one needs to implement both the barrel shifters and the Processing Elements. Fortunately, there are two newly-available fast LSI chips which combine a 16-bit ALU with a number of registers and a barrel shifter; these being the AMD 29116 and the TMS 3020. These chips are reasonably good for this application since each one bit section of the ALU/Register/Data Path can function as a one-bit Processing Element as well as providing the parallel shifting capability. Thus we get 16 of our 1-bit processors and a row (or column) barrel shifter on one chip. Without access to the appropriate VLSI fabrication facilities needed for a 32 x 32 DisArray2, we could use one of these two components for a 16 x 16 version.

The AMD 29116 is a 100ns component, and using this has the added benefit that the 16-bit ALU can also carry out 16-bit arithmetic in 100ns. This means that with 32 of these processors in the 16-vertical, 16-horizontal arrangement, the array can achieve all that the DisArray currently can, together with faster shifting and having a capability of over 300 MIPs arithmetic processing performance on arrays of 16-bit quantities.

There are a number of important graphics algorithms which would have a greatly improved performance on this type of hardware. Some of these are :

1. Line drawing with a DDA algorithm. The 16 horizontal (or vertical) 16-bit processors can work together to plot up to 16 points simultaneously from an arbitrary line across a 16 x 16 plane.
2. 16 of the processors can co-operate in polygon filling applications.
3. Co-ordinate transformation of multiple data points can be achieved in parallel with well-thought out arrangements of data.
4. If a bit-reversal path could be included in each of the processors (which entails designing the processor chip from start), it is possible to mirror bitmaps and transpose matrices. Amongst other cases, this can be used to rapidly calculate transitive closure relationships which are very useful in advanced window-based interactive systems (Cook (6)).

13.10.4 Other DisArray2 Features

In addition to the above, DisArray2 has a number of other features which can only be mentioned briefly here :

1. The Address staggering scheme of DisArray for video refresh.

2. A second similarly-arranged video shift register for the real-time input of video data.
3. The quadrant addressing scheme to speed up operations on neighbouring planes.
4. There will be two fast RAMs at each intersection point in the array, one being associated with the horizontal processor and one with the vertical processor. With this arrangement and a suitable mapping of array storage onto the address space of the host processor, it will be possible to fully utilise the processing capability of both sets of processors for well-organised vector arithmetic; for co-ordinate transformations, say. This would ensure that the 300 MIPs rating would not be degraded by both sets of processors competing for the same memory (in the cases where memory allocation could be so organised).

13.11 CONCLUSIONS

One solution to the problem of increasing the speed of the RasterOp graphics primitive has been presented, which uses a regular two-dimensional array of simple processor/memory pairs. A working version of this architecture has been built and performs its task well.

Further work may increase the speed still further in a later version of the hardware. Hopefully, this later version will be in the form of a powerful personal computer fully integrated with a more general-purpose array processor than the current one. This combination would provide a very useful amount of computing power for the execution of graphics and other applications.

This hardware was only designed to solve just one of the problems of high-performance graphics; namely that of executing RasterOp at high speed. However, it has come to be appreciated that this hardware architecture, suitable enhanced, will also be capable of performing a very significant amount of the number-crunching required for a more general-purpose graphics environment. We have strong hopes that the architecture will be of some benefit in the parallel execution of functional languages. Also, because of its great regularity, the architecture is eminently suitable for VLSI implementation. The possibility of an 'intelligent' RAM chip for this and other applications is very attractive.

A large part of the construction and de-bugging of the prototype has been undertaken by Mr. Jayesh Khatri, who also designed the array control unit. The micro-code implementation of RasterOp and line drawing by pre-computed strokes was done by Mr. Mohammed Ajab who also implemented a micro-assembler system for the array control unit. The author gratefully acknowledges the work of both of the above research assistants and also the support of the United Kingdom Science and Engineering Research Council and International Computers Ltd. in this work.

13.12 REFERENCES

1. Newman and Sproull, 1979, 'Principles of Interactive Computer Graphics', 2nd Edition, McGraw-Hill.
2. Thacker C.P. et al., 1979, 'Alto : A Personal Computer', Xerox Palo Alto Research Center Report.
3. Three Rivers Computer Corporation, 'Perq Software Reference Manual', 720 Gross St., Pittsburgh, PA.
4. Reddaway S.F., 1973, 'DAP - A Distributed Array Processor', 1st. annual Symposium on Computer Architecture, Gainesville, Florida.
5. L. Guibas, J. Stolfl, 'A Language for BitMap Manipulation', ACM Transactions on Graphics, Vol. 1, No. 3.
6. Cook S., 'Playing Cards', Software Practice and Experience, 13, 1043-1053.
7. Goldberg and Robson, 1983. 'Smalltalk-80, The Language and its Implementation', Addison-Wesley.
8. Page I. and Walsby A., 1978, 'The Q.M.C. Text Terminal', Electronic Displays '78, Conference Proceedings, London.
9. Gupta S., 1981, 'Architectures and Algorithms for Parallel Updates of Raster Scan Displays', Carnegie-Mellon University Report CMU-CS-82-111.

Directions in functional programming research

S. L. Peyton-Jones

14.1 INTRODUCTION

In 1968, Dijkstra's famous letter [Dijks68] proposed the view that the **goto** statement was a harmful feature of programming languages. In due course, this gave rise to a new class of **structured programming languages**, which lacked the **goto** statement, but which in compensation provided a set of **control structures**.

In 1977, Backus's Turing award lecture [Back78] proposed the view that the **assignment** statement was a harmful feature of programming languages. This view (which has early origins; see for example [Land66]) has given rise to the class of **functional programming languages**, which lack the notion of **assignment** (and hence **side-effects**), but which in compensation support **functions as first class citizens** (that is, functions may be passed as arguments to functions, returned as results, stored in data structures and so on).

There are two principal reasons for supposing the absence of the assignment statement to be a good thing:

- (i) The absence of side effects leads to clear and simple semantics, which makes programs **easier to write**, and **easier to reason about** than with **conventional languages**.
- (ii) Distinct sub-expressions of a program can safely be **evaluated concurrently**, since the absence of side effects ensures that the sub-expressions are genuinely independent. This opens up possibilities for the exploitation of parallel hardware.

In addition, and perhaps somewhat surprisingly, two other important concepts have been developed almost entirely within the functional programming research community. These are **polymorphic typing** and **lazy evaluation**.

Functional languages are members of the class of **declarative languages**, that is, languages in which programs have a **declarative reading** which asserts properties of the

functions defined by the program, as well as an **algorithmic reading** which describes how the functions are to be computed. Another common term for functional languages is **applicative languages**.

A short introduction to functional programming is given by Turner in [Turn82], while Burge gives a fuller treatment [Burge75]. For the sake of definiteness, examples given in this paper will be written in the language KRC [Hamm84]. Function application is denoted by juxtaposition, thus $(f x)$.

This paper briefly reviews the present state of affairs in the field of functional programming, and outlines some of the major challenges which are now being addressed by current research. An attempt at a comprehensive survey would be too long or too superficial, so the content of the paper reflects the author's interests and prejudices. The rest of the paper is in four main sections, covering languages, program transformation, evaluation order, and implementations.

Some material beyond the scope of this paper is briefly alluded to; in particular, Scott's theory of domains [Scott81] [Scott82], and the lambda calculus [Hend80] [Baren81].

14.2 LANGUAGES

The work that has been done by a number of independent teams on functional programming language design seems to have produced a considerable convergence of opinion, which we will briefly review.

The language ML, originally designed as part of the Edinburgh LCF project [Gord79], is now being significantly redesigned to take account of the experience gained with ML and other functional programming languages [Miln83]; the new language will be called Standard ML.

Hope, designed by Burstall, MacQueen and Sannella [Burst81] at Edinburgh, is the language which Darlington's team at Imperial College are using for ALICE (see section 14.5). It is a large language (for instance, it supports user-defined operators which may be prefix, infix or distfix).

Turner has designed a series of languages, Sasl [Turn76], KRC [Turn82] [Hamm84], and now Miranda (which is under construction).

Lispkit, designed by Henderson [Hend80], is a functional language which shares the syntax and simplicity of Lisp, and is explicitly designed for easy portability.

14.2.1 Polymorphic Typing

Strong typing has been widely accepted in conventional programming practice as a technique which catches a large class of programming errors at the compilation stage. Unfortunately, strong typing sometimes leads to tiresome bureaucracy, and this problem becomes particularly serious in a functional programming style. Consider, for example, a function designed to compute the length of a linked list. The function will work equally well on a **list of integers** and on a **list of characters**, but a conventional strong typing system will force the programmer to write separate functions for these two data types.

The ability to write such generic functions is so important to functional programming that early functional programming languages were typeless. Fortunately, however, based on the work of Hindley [Hindl69], Milner developed the concept of **polymorphic typing** [Miln78] [Damas82], which generalises the notion of type to allow types such as **list of ***, meaning "a list of objects of any type". Now the length function has a well behaved type:

```
(list *) -> integer
```

Likewise, a function which reverses a list has type:

```
(list *) -> (list *)
```

Polymorphic typing thus gives us the best of both worlds - we can have complete type checking of fully generic programs. Milner also shows that the type of an expression can be **deduced from the source program by the compiler**, rather than explicitly declared by the programmer. This is the approach taken in ML, the first language to include polymorphic typing.

Hope also includes a polymorphic typing system, but it also allows **overloading** of operators, and this turns out to require the programmer to declare her types explicitly.

Miranda deduces its types like ML, but allows the programmer optionally to declare her types; if she does so, the compiler will check them.

An even more general polymorphic type system is exemplified in the language Ponder [Fairb82]. Another approach is given by Welch and Ellis, in [Welch83].

14.2.2 Syntax

The (concrete) syntax of programming languages is important to their comprehensibility, and there is increasing acceptance of two new syntactic constructs introduced by some functional languages: **pattern matching** and **Zermelo-**

Frankel set notation.

14.2.2.1 Pattern matching. Consider the function

```
fac n = if (n=0) then 1 else n * fac (n-1) fi
```

We could rewrite it in a more comprehensible way thus:

```
fac 0 = 1
fac n = n * fac (n-1)
```

and this method of defining functions by cases, depending on the structure or value of their argument(s) is called pattern matching. Notice that the order of the equations in this example is important - on the whole this seems to be undesirable, and in Hope, for instance, the order is unimportant. Here is an example in which the cases depend on the structure of the argument:

```
double [] = []
double (x:l) = 2*x : double l
```

where ":" is the infix "cons" operator, and "[]" denotes the empty list. It can be seen that (as a bonus) pattern matching also allows implicit selection of components of structures, eliminating the need for selector functions (like car and cdr).

Pattern matching is now present in almost all functional languages - for example, Standard ML includes it while the original ML did not to the same extent.

14.2.2.2 Set notation. Consider this program, which computes the infinite list of prime numbers:

```
primes = sieve [2..]
sieve (p:x) = p : sieve { n | n<-x; n%p > 0 }
```

where [2..] denotes the infinite list of integers beginning with 2, "<-" means "drawn from", and "%" is the remainder operator. Thus, the expression in curly brackets should be read "the list of all n, drawn from (the list) x, such that the remainder when n is divided by p is greater than zero".

This curly-bracket notation for describing a list, inspired by Zermelo-Frankel set notation, was first introduced in a functional language by Burstall and Darlington in NPL [Burst77], and taken up by KRC, Hope and Miranda. It is a declarative construct, and fits very gracefully into a functional context. There seems to be no doubt that it makes functional programs significantly shorter, although its acceptance is not yet as widespread as pattern matching.

14.2.3 Modules and Abstract Data Types

One substantial obstacle to the production of large functionally programmed systems has been the absence of system structuring tools, such as separate compilation facilities and abstract data types.

Functional languages are uniquely suited to supporting such facilities, since their support for higher order functions renders such modularity much easier. For example, an arithmetic package could take a prime number as argument and return a tuple of functions which were arithmetic operators modulo the prime.

Current functional languages have rather crude structuring facilities (Hope is perhaps the most advanced), but interesting proposals have been made by MacQueen [MacQ83] and Turner (in his language Miranda). Both of these proposals make some attempt to treat an **environment** (ie a binding of names to values) as a quasi-first-class object.

14.2.4 Functional and Logic Programming

The pure logic languages (of which Prolog [Clock81] is an impure variant) are also free from side effects and hence declarative [Kowal79]. However, the relationship between functional and logic languages is still extremely unclear.

Logic languages are based on the extremely powerful **unification** operation [Robin71] (a sort of two-way pattern match, by contrast with the one-way pattern matching of functional languages), which, among other things supports **backtracking** and, in principle, allows programs to be run **backwards** as well as forwards. However, logic languages are only **first order**, and it seems that most logic programs do not use the full power of the language (for example, few non-trivial logic programs are actually run backwards).

Darlington has proposed a method of extending functional languages to include unification [Dar183], by extending the **relative set abstraction** described in Section 14.2.2.2 to **absolute set abstraction**. Then, instead of writing

$$\{ x \mid x \leftarrow L; p \ x \}$$

meaning "the set of x drawn from L such that $(p \ x)$ ", we might write

$$\{ x \mid p \ x \}$$

meaning "the set of x such that $(p \ x)$ ", as one could in mathematics. As an example, consider

$$\text{split } s = \{ [s1, s2] \mid \text{append } [s1, s2] = s \}$$

where **append** is defined by

```
{A}  append [[], k2] = k2
{B}  append [(x:k1), k2] = x : append [k1, k2]
```

Now suppose $s = [1, 2]$ (so $s = 1:[2]$). We have

```
split s = { [s1, s2] | append[s1, s2] = 1:[2] }
```

and now we must perform a unification with the definition of `append`, to get (as one possibility, by unifying with {B})

```
split s = { [1:k1, k2] | append [k1, k2] = [2] }
```

Taking it one step further, unifying with {A}

```
split s = { [[1], [2]] | }
```

which is one valid result. The others will be generated from choosing different equations with which to unify.

Darlington shows that the technique will work for higher order functions as well, but there is a limit here, since unification is known to be undecidable at 2nd order and above [Huet73] [Huet75] [Goldf81] [Fages83]. This work is at an early stage but appears to have considerable potential.

14.2.5 Debugging

It is unclear whether debugging functional programs is harder or easier than debugging conventional programs. It seems clear that existing debugging techniques (like putting print statements in the suspected functions, or examining dumps) are inappropriate due to the absence of side effects, and the peculiar evaluation order caused by lazy evaluation.

On the other hand, the absence of side effects makes it easier to fully test functions in isolation, since no unexpected side effects can change the value returned by a function. Little work has been done on this problem, but Peyton Jones [Peyt83] reports on the experience of debugging a medium sized system.

14.3 PROGRAM TRANSFORMATION

One of the most significant claims of the functional language community is that functional programs are easier to reason about than imperative programs, due to the absence of side effects. There are various sorts of reasoning we might wish to perform, for example:

- (i) Transforming a comprehensible though inefficient **specification** into an efficient but obscure program [Dar181].

(ii) Designing **efficient data representations** [Dar180].

(iii) Proving **properties** of programs [Turn83].

This is an expanding field, and a rich source of references may be found in [Parts83]. We will confine ourselves here to a single example to give the flavour of the subject, taken from [Dar181]. Assume that the following functions are defined:

```
{lg.1} length [] = 0
{lg.2} length (x:s) = 1 + length s

{ap.1} append [] s2 = s2
{ap.2} append (x:s1) s2 = x : append s1 s2
```

and that we want to define a function to append two lists together and find the length of the resulting list (we use curly brackets on the left to identify equations). We can write it thus:

```
{lo2} lengthof2 s1 s2 = length (append s1 s2)
```

but while this definition is adequate and clear, it is somewhat inefficient, and we will derive a more efficient version. Instantiating {lo2} with $s1=[]$ gives

```
lengthof2 [] s2
= length (append [] s2)      instantiating by {lo2}
{lo2.1} = length s2          unfolding by {ap.1}
```

Now instantiating {lo2} with $s1=(x:s1)$ gives

```
lengthof2 (x:s) s2
= length (append (x:s1) s2)  instantiating by {lo2}
= length (x : append s1 s2)  unfolding by {ap.2}
= 1 + length (append s1 s2)  unfolding by {lg.2}
{lo2.2} = 1 + lengthof2 s1 s2  folding by ~{lo2}
```

Finally, we see that {lo2.1} and {lo2.2} constitute a complete new definition of lengthof2, which is "more efficient" than the first version.

14.3.1 Correctness

The correctness of these manipulations clearly depends absolutely on being able to freely replace left hand sides of definitions by their right hand sides, and vice versa (these manipulations are conventionally called **unfolding** and **folding** respectively). This is valid in a functional language, and totally invalid in a conventional imperative one (such as Ada or Pascal) since the meaning of a program fragment depends on its context. The context independence of functional programs, whereby the value of an expression depends only on the values of its sub-expressions, and not on the history of the computation, is called **referential**

transparency.

In fact, folding preserves only **partial correctness**, as can easily be seen if we fold the right hand side of the equation

$$f\ x = \dots \text{body} \dots$$

with itself, to get

$$f\ x = f\ x$$

whose evaluation does not terminate. Kott [Kott75] gives conditions for total correctness in the first order case (briefly, total correctness is preserved if more unfolds than folds are performed).

14.3.2 Transformation Systems

The proof is completely formal, and amenable to machine checking. However, it is not clear what "more efficient" means, so machine generation of proofs is problematic.

While in the example above the sequence of transformations applied is fairly simple, larger examples become considerably more complex, and there is no doubt that, even with machine assistance, it would become impossible to conduct large proofs one step at a time. There are two basic strategies for addressing this problem, which Partsch and Steinbruggen [Parts83] call the **generative set approach** and the **catalog approach**.

The generative set approach uses a basic set of elementary transformations (fold, unfold, instantiate and so on), together with some kind of **meta language** in which to express patterns of transformation, so that the proof can be conducted at a higher level of abstraction (cf [Gord79], where ML is used as a proof meta language, but in a different proof system). The proof then becomes a program in the meta language. Darlington is using Hope itself as a meta language for conducting proofs about Hope programs [Dar181].

The catalog approach works by proving, in advance, a sufficiently large set of powerful theorems about the (higher order) functions involved in the program. These theorems are, in effect, "high level transformations" and therefore much more tractable than the elementary transformations of the generative approach. The catalog approach works well when there is a very restricted set of higher order functions, such as in the language FP, where it leads to an "algebra of programs", as described by Backus [Back78]; further references are [Wad181] [Islam81] [Kieb81].

14.3.3 Specification Languages

There is no reason to restrict the language we are manipulating to be computationally feasible. For instance,

$$\text{sqrt } x = y \text{ such that } y \text{ is a real number and } y*y = x$$

is a perfectly reasonable specification for a square root function, and the process of program development can then be regarded as a (formal) transformation from the specification to a more efficient implementation. The specification language is then just an enriched version of the implementation language.

14.4 EVALUATION ORDER

A number of researchers are working on topics related to the **order of evaluation** of a functional program. This question raises both semantic and pragmatic issues, which we examine below.

14.4.1 Semantic Issues

14.4.1.1 Normal order reduction. The standard method by which functional programs (which are just expressions) are executed (or evaluated) is known as **reduction**. Consider, for example, the expression

$$(5 + 4) * (9 - 3)$$

There is more than one way of computing the value of this expression:

$$\begin{aligned} (5+4)*(9-3) &\Rightarrow 9*(9-3) \Rightarrow 9*6 \Rightarrow 54 \text{ or} \\ (5+4)*(9-3) &\Rightarrow (5+4)*6 \Rightarrow 9*6 \Rightarrow 54 \end{aligned}$$

Each step in this process is a reduction, and a subexpression which we can reduce (like $(5+4)$ or $(9-3)$) is called a **reducible expression**, or **redex**. A functional program imposes a **partial order** on the sequence of reductions to be performed (in contrast to imperative programs, which impose a **total order** on the statements to be executed), and so there may be many possible reduction orders, all of which will yield the same result (if they terminate).

It turns out that in a functional program, while all reduction orders yield the same result, they may not all involve the same amount of work; indeed, some of them may fail to terminate. Fortunately, Church and Rosser showed that one particular reduction order, called **normal order** was guaranteed to terminate if any order does so. The proof of this termination property, together with the property that all reduction orders give the same result (if they terminate), is the celebrated Church-Rosser theorem; proofs

of which vary from a few pages to several hundred (the shortest are [Welch75] and [Ross82]). The termination and correctness of other reduction orders is considered by Downey and Sethi [Down76].

14.4.1.2 Lazy evaluation. One of the most exciting developments in the functional programming area has been **lazy evaluation** [Hend76], which turns out to be a **direct consequence of normal order reduction**. Lazy evaluation allows us to describe and manipulate **infinite data structures** (without infinite cost). Such infinite data structures are a surprisingly useful programming technique, allowing a clean separation of data structure definition and use.

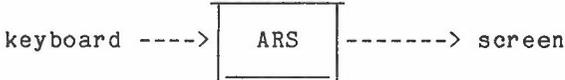
One important application of lazy evaluation is that it provides a very graceful way of integrating input and output (which seem rather side-effect laden) into a functional context. We may simply regard the keyboard input (for instance) as a potentially infinite list of characters (or **stream**), and likewise regard the result of the program (which is printed on the screen) as a stream. This approach is taken in Sasl, KRC, Miranda and Lispkit.

An example of another application of infinite data structures is arbitrary precision arithmetic, where the precision of the answer produced can be arbitrarily increased in a demand driven manner. One method of doing this is described by Peyton Jones [Peyt83a].

It has also been suggested by Bird [Bird84] that lazy evaluation can be used to give more efficient programs than could otherwise be written, even when operating on finite data.

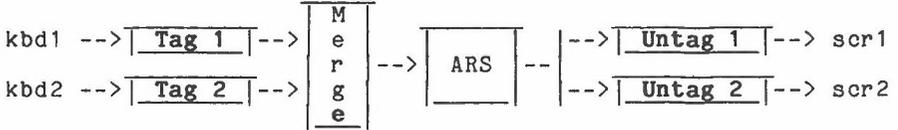
14.4.1.3 Nondeterminism. There seems to be growing agreement that some nondeterminism is necessary in a functional language that is to be used for systems programming, due to the nondeterminism inherent in input devices [Hend82].

When, for example, a program needs to respond to two input terminals, it needs to be able to process the "first" input which arrives. The easiest way of conceptualising this is to use a non-deterministic merge function, which interleaves the two input streams. For example, suppose we have an single-user airline reservation system, which is a function from a stream of input transactions to a stream of responses, thus:



screen = ARS keyboard

Now suppose that we want to make it a two-user system. The problem lies in the **arbitrary time sequencing** of the users' inputs. Using a nondeterministic merge primitive, we could proceed thus:



We first tag the two input streams, then merge them, feed them through the ARS function (now modified to carry through the tag to the output), and finally use the tag to steer the output to the appropriate screen. The code might look like this:

```

[scr1, scr2] = TwoUserARS [kbd1, kbd2]

TwoUserARS [kbd1, kbd2]
  = Split (ARS (Merge (Tag 1 kbd1) (Tag 2 kbd2)))

Split ARSOutput
  = [Untag 1 ARSOutput, Untag 2 ARSOutput]

Tag t stream = { [t,x] | x <- stream }
Untag t stream = { x | [x,tx] <- stream; tx = t }

```

Such a merge primitive should, ideally, have the following properties:

- (i) The result of (merge x y) should be **some interleaving** of the streams x and y ; that is, the result of (merge x y) contains only elements from x and y , with the relative order of each preserved.
- (ii) The merge should be \perp avoiding; that is

```

merge x  $\perp$  = x
and
merge  $\perp$  y = y

```

where we use the symbol \perp , pronounced "bottom", to denote the undefined value or non-termination.

- (iii) The merge should be **fair**; that is, all the elements of both x and y will (eventually) appear in the result. We note that a real user might want a stronger definition of fairness, since the definition

given would allow one input to be ignored for an arbitrarily long time, but this is a rather slippery concept to define formally [Park82].

Other workers in this area have used \perp avoiding **amb**, first introduced by McCarthy [McCar63], where $(\text{amb } a \text{ } b)$ is nondeterministically either a or b . Using **amb** it is possible to write a \perp avoiding, but not provably fair, merge (it is also, of course, possible to implement **amb** using merge).

Abramsky and Sykes [Abram82] describe an implementation with **amb** as the nondeterministic primitive, though it now directly supports a fair merge. Henderson and Jones [Jones83] describe a another implementation also based on **amb**.

The implementations of merge and **amb** both involve setting off two parallel processes to evaluate the two branches, and accepting as result the first to terminate. In both cases, however, there is a problem to do with terminating the process that is not chosen; we cannot forcibly terminate it, since other processes may by now be waiting for it. Neither implementation cited above attempts to garbage collect the processes thus generated, but when merging lists the effect of the uncollected processes is benign.

Nondeterminism significantly (but ineluctably) complicates the semantics of functional programs. Abramsky [Abram83] and Broy [Broy82] are working on the semantics of nondeterministic functional programs, using an approach based on powerdomains.

An alternative approach to the problem of nondeterminism is to make the time dependencies explicit by incorporating extra information in the data, and then write the program explicitly to use this information to resolve the scheduling choices. Variants of this approach include **hiatons** [Park82], **timestamps** [Broy83], and **oracles**.

14.4.1.4 Parallel conditionals. Consider the expression

if \perp then 3 else 3

If we attempt to evaluate this expression on a sequential machine, we will first evaluate the condition, which in this case does not terminate, so our evaluation of the expression will not terminate. However, it has a perfectly well defined value, viz 3. A slightly more complicated example is

if \perp then [3,4] else [3,5]

Here we can give some properties of the value of the expression (it is a list of two elements, whose first element is 3), but not others (we cannot tell whether the

second element of the list is 4 or 5). These examples suggests that our programming language would be more expressive if conditionals obeyed the following rules:

```

if True then x else y = x
if False then x else y = y
if ⊥ then x else y = GLB( x, y )

```

where $GLB(x, y)$ is the most defined object consistent with both x and y (or, in the Scott terminology, the greatest lower bound of x and y). This sort of conditional is called a **parallel conditional** [Fried78].

This rather complicated idea can be related to a simpler primitive, the **parallel OR**, which obeys the rules:

```

OR True x      = True           (for any x,
OR x True      = True           including ⊥)
OR False False = False

```

This primitive is not quite powerful enough, and it turns out that a parallel OR which operates on a **countable** (not just finite) set of disjuncts is sufficient to implement the parallel conditional. In fact, Plotkin shows that the addition of such a countable parallel OR to a language (called PCF), consisting of the typed lambda calculus together with arithmetic, makes the language powerful enough to express all the **computable functions** in its Scott domain [Plotk77]; a rather satisfying result. Scott proved a similar completeness result about his language LAMBDA, which includes the untyped lambda calculus [Scott76].

14.4.2 Pragmatic Issues

14.4.2.1 Graph reduction. While normal order reduction has good termination properties, it is rather expensive if implemented straightforwardly on acyclic parse trees, because it implements **call by name** semantics. Call by name involves, in effect, **textual substitution** of arguments in a function body. This is good because the evaluation of arguments is delayed until they are needed to produce the result of the function, but it is inefficient because it may involve repeated evaluation of the arguments.

Fortunately, normal order reduction has been rendered practical by a technique developed by Wadsworth [Wadsw71] known as **normal graph reduction**, which implements **call by need** semantics. Call by need is the same as call by name, except that it guarantees that no argument is evaluated more than once; thus arguments are evaluated only when needed, and then at most once. This effect is achieved by making all uses of an argument in a function body share a single pointer to the argument (hence the tree becomes a graph).

Turner [Turn79] describes a transformation \triangleright by which all variables are removed from the program, at the cost of

introducing some extra constant functions, called **combinators** [Curry58]. The (graph) reduction of the resulting expression is particularly simple, since it proceeds by smaller steps than Wadsworth's technique.

Implementations of these techniques are described in section 14.5.

14.4.2.2 Optimal reduction orders. Normal order always selects the **leftmost outermost** redex for the next reduction step. It has been known for some while that normal order reduction is not always optimal, in the sense of getting to the answer in fewest reduction steps, even if implemented by normal graph reduction [Wadsw71]. In fact, for the case of reduction by textual substitution, it is known that the optimal reduction order cannot be chosen by any computable strategy [Baren81]. No comparable result is known for graph reduction.

In the case of combinator reduction, however, normal order reduction is known to be optimal. This result, among many others on graph reduction, is shown in Staples' series of papers [Stap180] [Stap180a] [Stap180b]. A more accessible treatment of this work is given by Kennaway [Kenn84].

Sleep and Kennaway [Kenn84a] have suggested another reduction order, called **innermost spine reduction**, which enjoys the following properties:

- (i) It terminates if normal order does.
- (ii) It never takes more reduction steps than normal order.

Informally, this technique selects the innermost redex on the longest leftmost path from the root of the expression (the left **spine**). We may illustrate the difference between this and normal order by an example. Suppose

$$f = \lambda w.(w A (w B))$$

$$g = \lambda x.((\lambda y.(x y)) C)$$

Now consider evaluating $(f g)$. Normal order would produce:

$$\begin{aligned} f g &=> g A (g B) && \text{(one shared copy of } g) \\ &=> (\lambda y.(A y)) C (g B) \\ &=> A C (g B) \\ &=> A C ((\lambda y.(B y)) C) \\ &=> A C (B C) \end{aligned}$$

while innermost spine order produces:

$$\begin{aligned} f g &=> g A (g B) && \text{(one shared copy of } g) \\ &=> h A (h B) && \text{(where } h = \lambda x.(x C) \text{)} \\ &=> A C (h B) \\ &=> A C (B C) \end{aligned}$$

taking one fewer reduction to do so. The reason for this

saving is that innermost spine order reduces the redex inside the function *g* first of all, before applying *g* (which involves taking a copy of the body of *g*, thus duplicating any internal redexes, which will now have to be reduced in two separate places, as indeed happens in the normal order case).

It seems that, in practice, normal order reduction is "very nearly" optimal, and can only be improved on in carefully chosen examples. The practical impact of optimal reduction orders is therefore fairly limited. Levy is also working in this area [Levy80].

14.4.2.3 Space complexity. A more serious objection to normal order reduction has been raised by Hughes [Hugh84], which he illustrates with the following example.

Suppose we define

$$\text{AverageLineLength } L = (\text{Length } L) / (\text{Length } (\text{SelectCRs } L))$$

where *SelectCRs* returns a list of all the carriage return characters in its argument. Now suppose that *L* is the list of characters in a large file. If we wrote in Pascal, we could write a program which uses bounded space to compute the average line length, simply by maintaining a count of the number of characters so far, and the number of lines so far.

Unfortunately, a conventional functional language implementation will first evaluate one argument of the division operator and then evaluate the other. This means that the entire file will reside in memory at once, and the space usage is unbounded. It is clear that we would like to evaluate the arguments in parallel and in a synchronised fashion (notice that the first does not imply the second).

In the particular example given, it is possible to write a more efficient version without resorting to parallelism, but it is rather more obscure. More seriously, though, Hughes shows that there are simple and common programs which cannot run in bounded space on any sequential evaluator. This is not merely a theoretical problem; for example, one of the tools written for the Lispkit system is a screen editor, and it slowly uses up all the memory of the system as time goes on, until none remains and the system crashes.

Another example of the seriousness of this problem is the space complexity of a straightforward coding of the quicksort algorithm. It turns out that this has an average space complexity of $O(n)$ but a worst case space complexity of $O(n^2)$ (the imperative algorithm has a space complexity of $O(n)$).

Hughes therefore suggests that even on a single processor

implementation, some form of parallelism is essential if functional programs are to run efficiently. His proposed solution is to introduce two new language constructs, **par** and **synch**. The notation

$$f \text{ (par } x)$$

is semantically equivalent to

$$f \ x$$

but evaluates x in parallel with applying f to x . The value of the expression

$$\text{synch } e$$

is

$$\text{cons } e \ e$$

except that e will not be evaluated until BOTH the head AND the tail of $(\text{synch } e)$ are required. If, for example, the head is required before the tail, then the (parallel) process trying to evaluate the head will be suspended until another process tries to evaluate the tail, at which point both processes continue in parallel again. In the example given above, two parallel processes to compute $(\text{Length } L)$ and $(\text{Length } (\text{SelectCRs } L))$ may be resynchronised together whenever they consume a new element of L .

The way in which these constructs can be used to alleviate the space usage problem is too complex to describe here, but suffice it to say that the technique **does not alter the program's structure**. Even so, putting in the **par** and **synch** constructs in the right place is a subtle business, and if done incorrectly can cause the program to work less efficiently or even fail to terminate.

14.4.2.4 Strictness and parallelism. Functional languages are promoted as a natural vehicle for programming parallel machines, so it is legitimate to ask where such parallelism comes from.

It turns out that it all comes from evaluating the argument(s) of a function in parallel (with each other and with the function itself). For instance (in Hughes' notation)

$$f \text{ (par } x) \text{ (par } y)$$

will evaluate x and y in parallel, while simultaneously applying f to them. When, then, can we evaluate the argument of function in parallel? It can certainly never render our program incorrect (since it can have no side effects), which is reassuring, but it may not be a good way to use the machine's resources.

One approach is to drop the problem in the programmer's lap,

and insist that she annotate her program (as in the above example) to indicate where parallelism is desired. However, we might prefer to find some automatic technique for discovering when it is safe to evaluate an argument in parallel. In this context, it is "safe" to evaluate an argument to a function in parallel if we know that the function will need the value of its argument, and hence must eventually evaluate it. For instance, consider the function

$$f \ x \ y = \text{if } (\text{expensive } y) \text{ then } x+1 \text{ else } x+2$$

where "expensive" is some expensive function returning a boolean. We know that x 's value will be needed, and we could evaluate it in parallel. This kind of analysis is sometimes called **strictness analysis** (a function is **strict** if it always evaluates its argument).

Mycroft's thesis [Mycr81] describes a method for performing this strictness analysis. His method works by executing the program in a much reduced data domain (abstract interpretation). There is a classic example of the technique in the "rule of signs":

$$\begin{aligned} (+) * (+) &= (+) \\ (-) * (-) &= (+) \\ (+) * (-) &= (-) \\ (-) * (+) &= (-) \end{aligned}$$

Here we can discover something about the $*$ operator by executing it in a data domain reduced to $(+)$ (the positive numbers) and $(-)$ (the negative numbers). Mycroft suggests reducing the data domain to "undefined" and "defined". Then if, by executing the program in this reduced domain, we find that

$$f \ \text{"undefined"} = \text{"defined"}$$

then f does not evaluate its argument, so we must not evaluate the argument in parallel (because it may not terminate). If, on the other hand

$$f \ \text{"undefined"} = \text{"undefined"}$$

then f does evaluate its argument. In this case, it is safe to evaluate the argument in parallel. Matters are in fact more complicated than this, but only slightly, and Mycroft's work comes complete with a sound theoretical basis. However, his work only applies to flat domains and first order functions.

Other workers in this area are Johnsson [Johns81], Meira [Meira84] and Mishra [Mishr84]. The latter two are trying to extend the work to non-flat domains and higher order functions.

14.4.2.5 Strictness and sequential machines. One major source of inefficiency in many lazy sequential evaluators is the mechanism for postponing the evaluation of function arguments until their value is needed (often a **closure** must be built). However, if the function is known to be strict, its argument can be evaluated at call time, just as is the case with conventional call-by-value implementations of imperative languages. The technology for doing this (stacks in particular) is well understood and very efficient.

This was the original motivation for Mycroft's work, and is used in the G-machine compiler [Johns83], and the compiler of Hudak and Kranz [Hudak84] (see section 14.5).

14.5 IMPLEMENTATIONS

14.5.1 Reduction Machine Models

A considerable amount of work has been done in search of efficient methods for actually performing reductions on the program once the evaluation order has been determined. From a semantic point of view, performing a reduction involves replacing a function application by a copy of the function body with the function argument substituted for the bound variable. This may be implemented in one of two ways:

- (i) By actually copying the function body at the time of application, replacing occurrences of the bound variable with (pointers to) the argument. This is called **graph reduction**
- (ii) By a **delayed substitution** method, whereby the bound variable is associated with the argument in an **environment**, and when occurrences of a bound variable are subsequently encountered, they are dynamically looked up in the environment to determine the value associated with the variable.

A complete description of the various current implementations would take another paper, so we will give brief references to the main sources.

14.5.1.1 Graph reduction machines. The original description of a practical normal graph reduction technique is given by Wadsworth [Wadsw71]. Hughes [Hugh82]. describes a method, which he calls **supercombinators**, in which all free (ie unbound) variables of functions are made into explicit parameters. This, together with some other important optimisations, strictly limits the amount of copying that has to be done on any function application, but the method is basically similar to Wadsworth's. Variant of this technique are employed in the **G-machine**, an extremely fast Vax implementation of ML [Johns83], and in the compiler of Hudak and Kranz [Hudak84a].

Turner [Turn79] [Turn79a] describes a combinator reduction model, which forms the basis of his implementations of Sasl and Miranda. A combinator reducer has also been implemented in (microcoded) hardware [Clark80] [Stoye83] [Stoye84].

A multiprocessor machine called ALICE based on copying graph reduction is under construction at Imperial College [Dar181a]. This machine is intended to exploit the parallelism implicit in functional languages, and briefly alluded to earlier.

A totally different approach is taken by Mago [Mago79] [Mago80] whose machine is based on reduction by **textual substitution**, relying on massive parallelism to overcome the inefficiency caused by the copying involved (see section 14.4.2.1).

These and other implementations are reviewed by Treleaven in [Trel82] [Trel82a] [Trel83], which include comprehensive bibliographies.

14.5.1.2 Delayed substitution. The best-known implementations of an almost functional language via delayed substitution are, of course, those of Lisp (see, for example, [Sussm82]).

The other classic description of a delayed substitution model implemented via compilation (rather than interpretation) is the SECD machine, first described by Landin [Land63], and taken up by Henderson in his implementation of Lispkit [Hend80].

14.5.2 Memory System and Garbage Collection

In common with Lisp, all functional programming systems require a garbage collected heap, and there is well proven technology for managing such a heap (see [Cohen81] for a survey). However, two new factors are becoming important:

- (i) **Multiprocessor systems.** One of the advantages of functional languages stated in the introduction was the possibility of parallel multiprocessor implementations. This has important implications for storage management.
- (ii) As Multics showed, it is possible to incorporate a computer's filing system as part of its virtual memory space. In a functional context this would be particularly convenient, but would entail supporting a **very large heap** which was **persistent** from day to day. Such a large heap, of which only a small fraction will be in active use, needs careful management.

There are several well known techniques for providing storage management for a heap. Among these are **mark-scan**,

copying (Baker's algorithm [Baker78] in particular), and reference counting garbage collectors.

The mark-scan and copying algorithms have the following characteristics:

- (i) They require the **system to be stopped** while garbage collection takes place. This causes problems for
 - (a) **Critical real time systems**, for which such a pause may be unacceptable.
 - (b) **Parallel multiprocessor systems**, for which the global synchronisation of garbage collection may be difficult.

We would prefer a garbage collector which was distributed in time, and distributed in space (among the parallel processors). Baker's algorithm can, in fact, be distributed in time to some extent, but it is still impossible to guarantee that all storage requests can be satisfied immediately.

- (ii) They take **time proportional to the total amount of accessible data** to perform a collection. This might be very large, particularly if the heap persisted for longer than a single program run.
- (iii) They can reclaim **circular structures**.
- (iv) They have very **small storage requirements**.

The reference counting technique has different set of characteristics:

- (i) It is **distributed in space and time**, and reclaims store as soon as it becomes free. This also tends to lead to a smaller working set in virtual memory systems.
- (ii) It is not capable of reclaiming **circular data structures**.
- (iii) It has significant **extra storage requirements**.
- (iv) It **visits only objects currently being processed**, not all accessible objects.
- (v) It is **more awkward to use**, since the reference counts need to be explicitly updated.

14.5.2.1 Improving reference counting garbage collection.

Hughes [Hugh82a], has suggested an extension to the conventional reference counting algorithm that would allow it to reclaim circular structures.

The key idea is simple and elegant. We regard the accessible data in the heap as a **directed graph**, and to divide this graph into its **strongly connected components**.

In this context we recall that

- A graph is **strongly connected** if, for any two nodes A and B, there is a path from A to B, and vice versa.
- A **strongly connected component** of a graph is a maximal strongly connected subgraph.

Now, it is clear that

- (i) If one node of a strongly connected component is accessible, then all its nodes are (and vice versa).
- (ii) If we coalesce all the nodes in each strongly connected component, then the resulting **derived graph** is acyclic.

But now, since the derived graph is acyclic, it is amenable to conventional reference counting garbage collection; and when a node of the derived graph becomes unreferenced, then all the nodes of the corresponding strongly connected component have become unreferenced.

Hughes therefore suggests adding a reference count field to each node, which either contains the **shared reference count** for the strongly connected component of which the node is part, or is used to point at the node which does hold the shared reference count. He gives algorithms for incrementally maintaining the information of which components are strongly connected, and shows that they are rather cheap, except where a strongly connected component is split into two.

It appears that this technique can successfully alleviate the "circular data structure problem", and thus allow exploitation of the other desirable characteristics of reference counting, but no implementations yet exist. Brownbridge [Brown84] describes a different reference counting technique, also capable of recovering circular structures.

14.5.2.2 Exploiting cell lifetimes. Another approach recently suggested by Hewitt and Lieberman [Lieb83] is based on the observation that

The longer a cell has lived,
the longer it is likely to live.

Consider, for example, a heap which includes a filing system. Many files will be unused for long periods, while data structures that are currently being processed will have relatively short lifetimes. A conventional copying collector will copy the entire filing system each time it runs - a very wasteful activity, since it is unlikely to recover any space from the inactive majority of the filing system.

Hewitt and Lieberman therefore suggest dividing the address

space into **regions** of increasing age. Most pointers point backwards in time (that is, if they cross region boundaries, they will mostly point from younger regions to older ones). Where pointers point from an older region into a younger one (only update operations can cause this), they are constrained to go via an **entry table** associated with the younger region.

Now the youngest region can be garbage collected **independently**, using Baker's algorithm, so long as we **preserve all cells referenced from its entry table**. In general, any region can be garbage collected without touching any older information. So all we have to do is to garbage collect young regions (where garbage collection will be fruitful), more often than older ones (where it will be less fruitful, but eventually necessary).

At the time of writing of their paper, Hewitt and Lieberman did not appear to have implemented their proposal.

14.5.2.3 Avoiding garbage collection. Another approach to garbage collection is to try to avoid it altogether. Wadler [Wad184] suggests a technique for compiling a certain class of functional program into a **finite state machine** with a fixed number of registers and no heap. This, in effect, performs memory allocation in advance (rather as a conventional Pascal program has no problem with memory allocation). He calls his compiler **the listless transformer**.

The functional programs to which this technique is applicable are, not surprisingly, those which can be evaluated using **bounded internal storage**. This includes, for example, functions which find the length of a list, add up a list, appending or merging two lists, dividing a list into two lists of odd and even elements. It excludes, however, functions which sort a list, append a list to itself, or which work on tree-shaped data.

Clearly the applicability of the method is limited, but where appropriate it is extremely effective, since the finite state machine can be made very fast.

Wadler has a working implementation of his listless transformer, written in KRC.

14.6 ACKNOWLEDGEMENTS

A number of people have provided ideas and feedback which have significantly improved this paper, including Samson Abramsky, Sue Astley, John Darlington, John Hughes, David Park, David Turner, Ronan Sleep, Phil Wadler, and John Washbrook.

References

- [Abram82] Abramsky S, "SECD-M - a virtual machine for applicative multiprogramming", Computer Systems Lab, Queen Mary College, Nov 1982.
- [Abram83] Abramsky S, "On semantic foundations for applicative multiprogramming", Computer systems lab, Queen Mary College, 1983.
- [Back78] Backus J, "Can programming be liberated from the von Neumann style?", CACM 21(8), pp613-641, Aug 1978.
- [Baker78] Baker H, "List processing in real time on a serial computer", CACM 21(4), pp280-294, April 1978.
- [Baren81] Barendregt HP, "The lambda calculus, its syntax and semantics", North Holland, 1981.
- [Bird84] Bird RS, "Using circular programs to eliminate multiple traversals of data", Programming research group, Oxford, 1984.
- [Brown84] Brownbridge D, "Recursive structures in computer systems", PhD thesis, University of Newcastle on Tyne, 1984.
- [Broy82] Broy M, "A fixpoint approach to applicative multiprogramming", in Theoretical foundations of programming methodology, ed Broy and Schmidt, D Reidel, 1982.
- [Broy83] Broy M, "Applicative real-time programming", Proc 9th IFIP, Information Processing 1983, North Holland, pp259-264, 1983.
- [Burge75] Burge WH, "Recursive programming techniques", Addison Wesley, 1975.
- [Burst77] Burstall, RM, "Design considerations for a functional programming language", Proc Infotech State of the Art Conference, Copenhagen, 1977.
- [Burst81] Burstall RM, MacQueen DB, Sannella DT, "Hope - an experimental applicative language", Edinburgh report CSR-62-80, 1981.

- [Clark80] Clarke TJW, Gladstone PJS, Maclean CD, Norman AC, "SKIM - the S K I reduction machine", Proc ACM Lisp Conference, Stanford, 1980.
- [Clock81] Clocksin and Mellish CS, "Programming in Prolog", Springer Verlag, 1981.
- [Cohen81] Cohen J, "Garbage collection of linked data structures", ACM Computing Surveys 13(3), pp341-367, Sept 1981.
- [Curry58] Curry HB and Feys R, "Combinatory Logic, Vol 1", North Holland, 1958.
- [Damas82] Damas L and Milner R, "Principal type schemes for functional programs", Proc ACM Symposium on Principal of Programming Languages, pp207-212, 1982.
- [Dar180] Darlington J, "The design of efficient data representations", Imperial College, 1980.
- [Dar181] Darlington J, "The structured description of algorithm derivation", in Algorithmic Languages, ed de Bakker and von Vliet, North Holland, 1981.
- [Dar181a] Darlington J and Reeve M, "Alice - a multiprocessor reduction machine for the parallel evaluation of applicative languages", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp65-75, Oct 1981.
- [Dar183] Darlington J, "Unifying logic and functional languages", Imperial College, 1983.
- [Dijks68] Dijkstra EW, "The GOTO statement considered harmful", CACM 11(3), pp147-148, March 1968.
- [Down76] Downey PJ and Sethi R, "Correct computation rules for recursive languages", SIAM Journal of Computing 5(3), pp378-401, Sept 1976.
- [Fages83] Fages F and Huet GP, "Complete sets of unifiers and matches in equational theories", Proc 8th Colloquium on Trees in Algebra and Programming, Springer Verlag LNCS 159, pp205-220, 1983.
- [Fairb82] Fairbairn J, "PONDER, and its type system", Cambridge Computer Lab Technical Report 31, 1982.

- [Fried78] Friedman DP and Wise DS, "A note on conditional expressions", CACM 21(11), pp931-933, Nov 1978.
- [Goldf81] Goldfarb W, "The undecidability of the second order unification problem", Theoretical Computer Science 13, pp225-230, 1981.
- [Gord79] Gordon MJC, Milner AJ, Wadsworth CP, "Edinburgh LCF", Springer Verlag LNCS 78, 1979.
- [Hamm84] Hammond K, "The KRC manual", CSA/16/1984, DSAG-3, University of East Anglia, May 1984.
- [Hend76] Henderson P, Morris J, "A lazy evaluator", Proc ACM Symposium on the Principles of Programming Languages, 1976.
- [Hend80] Henderson P, "Functional programming", Prentice Hall, 1980.
- [Hend82] Henderson P, "Purely functional operating systems", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp177-192, 1982.
- [Hind169] Hindley R, "The principal type scheme of an object in combinatory logic", Trans American Mathematical Society 146, pp29-60, 1969.
- [Hudak84] Hudak P and Kranz D, "A combinator based compiler for a functional language", 11th Symposium on Principles of Programming Languages, pp122-132, Jan 1984.
- [Hudak84a] Hudak P and Kranz D, "A combinator based compiler for a functional language", Eleventh Symposium on Principles of Programming Languages, pp122-132, Jan 1984.
- [Huet73] Huet GP, "The undecidability of unification in third order logic", Information and Control 22, pp257-267, 1973.
- [Huet75] Huet GP, "Unification in the typed lambda calculus", Proc Symposium on the lambda calculus and computer science theory, Springer Verlag LNCS 37, pp192-212, 1975.
- [Hugh82] Hughes RJM, "Graph reduction with supercombinators", PRG-28, Programming research group, Oxford, June 1982.

- [Hugh82a] Hughes RJM, "Reference counting with circular structures in virtual memory applicative systems", Programming research group, Oxford, 1982.
- [Hugh84] Hughes RJM, "Parallel functional programs use less space", Programming research group, Oxford, 1984.
- [Islam81] Islam N, Myers TJ, Broome P, "A simple optimiser for FP-like languages", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp33-40, Oct 1981.
- [Johns81] Johnsson T, "Detecting when call by value can be used instead of call by need", LPM Memo 14, Chalmers Inst, Sweden, Oct 1981.
- [Johns83] Johnsson T, "The G-machine", Proc Workshop on Declarative Programming, University College London, Apr 1983.
- [Jones83] Jones S, "Abstract machine support for purely functional operating systems", PRG-34, Programming research group, Oxford, Aug 1983.
- [Kenn84] Kennaway JR, "An outline of some results of Staples on optimal reduction orders in replacement systems", University of East Anglia, Mar 1984.
- [Kenn84a] Kennaway R, "Private communication", , April 1984.
- [Kieb81] Kieburtz RB, Shultis J, "Transformations of FP program schemes", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp41-48, Oct 1981.
- [Kott75] Kott L, "About a transformation system - a theoretical study", Proc 3rd Symposium on Programming, Paris, 1975.
- [Kowal79] Kowalski R, "Logic for problem solving", North Holland, 1979.
- [Land63] Landin PJ, "The mechanical evaluation of expressions", Computer Journal 6(4), pp308-320, 1963.

- [Land66] Landin PJ, "The next 700 programming languages", CACM 9(3), pp157-166, Mar 1966.
- [Levy80] Levy JJ, "Optimal reductions in the lambda calculus", in *Essays on combinatory logic, lambda calculus and formalism*, ed Hindley and Seldin, Academic Press, pp159-192, 1980.
- [Lieb83] Lieberman H and Hewitt C, "A real time garbage collector based on the lifetimes of objects", CACM 26(6), pp419-429, June 1983.
- [MacQ83] MacQueen D, "Modules for SML", *Polymorphism* 1(3), Dec 1983.
- [Mago79] Mago GA, "A network of microprocessors to execute reduction languages", Two parts, *International Journal of Computer and Information Sciences* 8(5) pp349-385; 8(6) pp435-471, 1979.
- [Mago80] Mago GA, "A cellular computer architecture for functional programming", *IEEE Computer Society COMPCON*, pp179-187, 1980.
- [McCar63] McCarthy J, "A basis for a mathematical theory of computation", in *Computer Programming and Formal Systems*, ed Braffort and Hirschberg, North Holland, pp33-70, 1963.
- [Meira84] Meira S, "Optimised combinatoric code for applicative language implementation", *Proc 6th International Symposium on programming*, 1984.
- [Miln78] Milner R, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences* 17(3), pp348-375, Dec 1978.
- [Miln83] Milner R, "A proposal for Standard ML", *Polymorphism* 1(3), Dec 1983.
- [Mishr84] Mishra P and Keller RM, "Static inference of properties of applicative programs", *11th ACM Symposium on Principles of Programming Languages*, pp235-244, Jan 1984.
- [Mycr81] Mycroft A, "Abstract interpretation and optimising transformations for applicative programs", *Edinburgh CST-15-81*, 1981.
- [Park82] Park D, "The fairness problem and nondeterminism in computing networks", *Proc 4th Advanced Course on Theoretical Computer Science*, Mathematisch Centrum, 1982.

- [Parts83] Partsch H and Steinbruggen R, "Program transformation systems", ACM Computing Surveys 15(3), pp199-236, Sept 1983.
- [Peyt83] Peyton Jones SL, "Yacc in Sasl", Indra note 1533, University College London, Dec 1983.
- [Peyt83a] Peyton Jones SL, "Arbitrary precision arithmetic using continued fractions", Indra Note 1530, University College London, Dec 1983.
- [Plotk77] Plotkin G, "LCF considered as a programming language", Theoretical Computer Science 5, pp223-255, 1977.
- [Robin71] Robinson JA, "The unification computation", in Machine Intelligence 6, Edinburgh University Press, pp63-72, 1971.
- [Ross82] Rosser JB, "Highlights of the history of the lambda calculus", Proc ACM Symposium on Lisp and Functional Programming, Pittsburgh, pp216-225, Aug 1982.
- [Scott76] Scott D, "Data types as lattices", SIAM Journal of Computing 5, pp522-587, 1976.
- [Scott81] Scott D, "Lectures on a mathematical theory of computation", PRG-19, Programming research group, Oxford, May 1981.
- [Scott82] Scott D, "Domains for denotational semantics", 9th Colloquium on Automata, Languages and Programming, Springer Verlag LNCS 140, pp577-613, July 1982.
- [Stapl80] Staples J, "Computation on graph-like expressions", Theoretical Computer Science 10, pp171-185, 1980.
- [Stapl80a] Staples J, "Optimal evaluations of graph-like expressions", Theoretical Computer Science 10, pp297-316, 1980.
- [Stapl80b] Staples J, "Speeding up subtree replacement systems", Theoretical Computer Science 11, pp39-47, 1980.
- [Stoye83] Stoye WR, "The SKIM microprogrammer's guide", Cambridge Computer Lab Technical Report 40, Sept 1983.

- [Stoye84] Stoye WR, Clarke TJW, Norman AC, "Some practical methods for rapid combinator reduction", Proc ACM Symposium on Lisp and Functional Programming, Austin, Aug 1984.
- [Sussm82] Sussman GJ, "Programming and implementing Lisp", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp29-72, 1982.
- [Trel82] Treleaven P, "Computer architecture for functional programming", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp281-306, 1982.
- [Trel82a] Treleaven P, Brownbridge D, Hopkins R, "Data driven and demand driven computer architecture", Computing Surveys 14(1), pp93-143, Mar 1982.
- [Trel83] Treleaven P, "Decentralised computer architectures for VLSI", in VLSI Architecture, Prentice Hall, pp348-380, 1983.
- [Turn76] Turner DA, "The Sasl manual", St Andrews, Dec 1976.
- [Turn79] Turner DA, "A new implementation technique for applicative languages", Software Practice and Experience 9, pp31-49, 1979.
- [Turn79a] Turner DA, "Another algorithm for bracket abstraction", Journal of Symbolic Logic, 44(2), pp267-270, June 1979.
- [Turn82] Turner DA, "Recursion equations as a programming language", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp1-28, 1982.
- [Turn83] Turner DA, "Functional programming and proofs of program correctness", in Tools and Notions for Program Construction, Neel (ed), CUP, 1983.
- [Wad181] Wadler P, "Applicative style programming, program transformation, and list operators", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp25-32, Oct 1981.

- [Wad184] Wadler P, "Listlessness is better than laziness", Proc ACM Symposium on Lisp and Functional Programming, Austin, 1984.
- [Wadsw71] Wadsworth CP, "Semantics and pragmatics of the lambda calculus", D Phil dissertation, Oxford, 1971.
- [Welch75] Welch P, "Some notes on the Martin-Lof proof of the Church-Rosser theorem, as rediscovered by Park", Computer Lab, University of Kent, Oct 1975.
- [Welch83] Welch PH and Ellis MP, "Strong typing and functional programming", Computer Lab, Univ of Kent, 1983.

The Zero Assignment Parallel Processor (ZAPP) project

M. R. Sleep and J. R. Kennaway

15.1 INTRODUCTION

The original aim of the ZAPP project was to demonstrate that it is possible to "buy speed" for a significant class of problems by hooking together a large number of computing elements. For a suitable class of problems, we wanted to show that doubling the number of elements would roughly halve the run-time. At the same time, we wanted to maintain programmability so as not to further aggravate the software crisis.

The ZAPP goal transcends any particular technology such as VLSI because it asks "how can we exploit parallel hookups of the most advanced computing element offered by a particular technology". Today, we have single chip computers (e.g. the INMOS transputer). Soon we will have the ability to use WSI (Wafer Scale Integration) to create much more powerful computing devices. In due course we can expect even more powerful computing elements based on new technologies to emerge. The basic ZAPP question of how to buy speed from parallelism will always be relevant.

To sharpen this difficult question, we made the deliberate decision to restrict our attention to "divide and conquer" algorithms. We dealt with the question of programmability by adopting a functional (zero assignment) language. In a functional language, the evaluation of sub-expressions has no side effects so that parallel execution cannot produce a wrong answer, although an injudicious choice of ordering of subexpressions for execution can lead to undesirable consequences - e.g. nontermination or unnecessary saturation of system queues.

We cannot hope to "buy speed" for all problems. For example, to evaluate $x^{(2^n)}$ we cannot improve on sequentially squaring x , n times. At the other extreme, certain algorithms can be "systolicised", and special purpose chips will then yield very high performance, as described by Kung (1). We seek a basis for a more general purpose parallel architecture which makes economic use of

the available computational resources without requiring the programmer to delve too deeply into the physical architecture. This requires a good language-architecture symbiosis.

Chapters 7 and 8 of Kennaway and Sleep (2) give one overview of the increasingly active search for such a symbiosis. Here we present in condensed form some early results of the ZAPP approach. Further details are available in Burton and Huntbach (3).

15.2 ARCHITECTURE RESULTS

15.2.1 The Reduction Model of Computation

ZAPP is based on a simple reduction model of computation. A computational task is given by an expression E and a set of rewrite rules R (also called reduction rules). The rules are applied to replace subexpressions of E by other expressions. When E has reached a form to which no rewrite rule is applicable, this form is the result of the computation and is called a normal form. Arithmetic provides a simple example. The arithmetic operators are defined by a collection of rules such as $1+1 \rightarrow 2$, $2*3 \rightarrow 6$, etc. Given an expression

$$(3*4) + (5*6)$$

we can apply the rules

$$\begin{array}{ll} 3*4 & \rightarrow 12 \\ 5*6 & \rightarrow 30 \end{array}$$

to produce the intermediate (but equivalent) form

$$12+30$$

which may be reduced to 42 by the rule

$$12+30 \rightarrow 42$$

42 cannot be further reduced and is the result of the computation. Notice that the two multiplication reductions we applied to the initial expression could have been applied in any order or in parallel without affecting the result. This property, that the final result is independent of the order in which reductions are performed, is called the Church-Rosser property (Barendregt (4)). Rewrite systems possessing this property are particularly amenable to implementation on a parallel machine. When an opportunity to apply a rule appears it can be taken advantage of immediately - the final result is independent of the order of execution. The usual von-Neumann programming languages do not have this property. It takes a considerable effort to extract parallelism from an arbitrary FORTRAN program.

The classical example of a Church-Rosser rewrite system is the lambda calculus (4), which forms the core of all functional languages. With the addition of basic constants and operators, and syntactic sugaring, we obtain from it languages such as ISWIM (Landin (5)), ML (Gordon et al (6)), SASL (Turner (7)), HOPE (Burstall et al (8)), LISPKIT (Henderson et al(9)), etc. (For our purposes we ignore the support of assignment in ML and ISWIM.)

Using such languages involves writing equations which have a dual interpretation. On the one hand, each equation has a purely declarative reading - the two sides of the equation are asserted to be equivalent. On the other, each equation may be used by the architecture as a rewrite rule to simplify an expression. This is the procedural reading, and the (usually unwritten) meta-rule adopted by most architectures is to use equations only from left to right.

Kowalski (10) and others have developed such a dual reading for languages based on logic formalisms. The essential difference between logic and functional (lambda-calculus based) languages is (for architects) the fact that supporting logic requires a backtracking mechanism whereas supporting function languages does not. To keep things as simple as possible, early ZAPP work restricted attention to functional languages.

15.2.2 Process Trees

Divide and conquer algorithms are particularly amenable to parallel execution. A divide and conquer algorithm splits a problem into a number of subproblems, applies itself recursively to these subtasks, and then combines the results. As an example we give a divide and conquer algorithm for factorial:

```
DEF fac n = dfac l n
      WHERE dfac lo hi =
            IF lo=hi
            THEN lo
            ELSE (dfac lo mid) * (dfac mid+1 hi)
                 WHERE mid = (lo+hi) DIV 2
      FI
```

(dfac lo hi) gives the product of the numbers in the range lo..hi, and in the case where lo<hi is defined by splitting the range into two equal subranges, applying dfac to each, and multiplying the results.

Assuming an arbitrary element of our parallel architecture has a basic knowledge of the arithmetic and logical operators we could send (fac 4) together with the above definition of fac to some element and request its evaluation. The processing element receiving this task will first use the definition of fac to reduce the expression to:

```
dfac 1 4 WHERE dfac...
```

It can now use the definition of dfac specified in the WHERE-clause to produce the intermediate form:

```
( (dfac 1 mid) * (dfac (mid+1) 4)
  WHERE mid = (1+4) DIV 2 )
  WHERE dfac lo hi = ...
```

This expression is at top level a multiplication. Multiplication is a strict operator - that is, both its arguments must be fully evaluated before the multiplication can be performed. We therefore have the two subtasks

```
?1 = ( (dfac 1 mid) WHERE mid = (1+4) DIV 2 )
      WHERE dfac lo hi = ...
?2 = ( (dfac (mid+1) 4) WHERE mid = (1+4) DIV 2 )
      WHERE dfac lo hi = ...
```

and a top-level task which can be represented as an object with two "holes" waiting for the results of ?1 and ?2:

?1	*	?2
----	---	----

In principle, we could ship the two subtasks to neighbouring processing elements and await the results. When both are available, the top-level task will have the form $? * 12$ which can be reduced to the normal form 24. In this simple example, we have deliberately emphasised the need to communicate closures (pairs of the form (expression, definitions)) rather than simple expressions in a distributed architecture. It would obviously pay us to evaluate $mid = (1+4) DIV 2 = 2$ before splitting the top-level task. If we did this, the two subtasks would be

```
?1 = dfac 1 2 WHERE dfac lo hi = ...
?2 = dfac (2+1) 4 WHERE dfac lo hi = ...
```

We can achieve something close to this effect by copying references to shared subexpressions when we split a task. This works well for mid, but not so well for dfac itself: we would like every computing element to have its own copy of dfac to avoid a bottleneck. In a distributed implementation it will sometimes pay to make full copies rather than share.

Whatever the detailed pragmatics, the evaluation of expressions can be viewed as generating a process tree which has as its root node the closure of the original expression, and as leaf nodes the primitive subexpressions which cannot be reduced. A process tree for fac 4 is illustrated in Fig.15.1. Clearly there are many ways for a physical architecture to traverse this process tree. A purely sequential solution would grow a stack of tasks depth-first. A highly concurrent architecture might adopt a breadth-first approach. The depth-first extreme is good because it limits

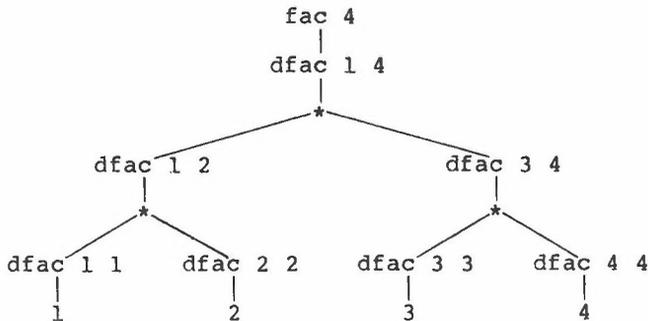


Figure 15.1 Process tree for fac 4

space requirements, but loses parallelism. The breadth-first approach looks good when an unbounded number of computing elements are available, but makes potentially huge space demands.

ZAPP architectures represent a compromise between the two extremes. Each ZAPP element adopts the conservative approach of using a sequential depth-first exploration of the process tree. This builds a stack of tasks on an individual ZAPP element. Parallelism is exploited by connecting a number of ZAPP elements together, and allowing underworked elements to "steal" tasks from the stacks of overworked physical neighbours. Each stolen task may generate subtasks which can themselves be stolen in turn. The effect is that work spreads over the network in a breadth-first way, while each processor executes its tasks depth-first.

15.2.3 ZAPP Architectures

ZAPP is not yet a physical architecture. At present, it exists only as simulations of an abstract set of rules which allow a single complex task to diffuse across any connected set of ZAPP elements in a manner which can make effective use of the available physical resources.

There are two sets of rules for an architecture to fall into the ZAPP class:

A. Interconnection. Any connected graph can be used to connect ZAPP elements, ranging from a simple ring to the r-n-cube (Burton and Sleep (11)). As suggested by Hillis (12) a given technology will dictate an appropriate interconnection scheme.

B. Behaviour of a ZAPP node.

(i) Each node of a ZAPP must be capable of simulating the parallel execution of a process tree using a depth-first priority rule.

(ii) Each node of a ZAPP must be capable of exchanging loading information at regular intervals with physical neighbours.

(iii) When underloaded, each element must be able to "steal" a task from any overloaded physical neighbour.

(iv) A task may be stolen at most once. This ensures that parent-child communication in the logical process tree involves at most one interconnection in the physical architecture. Somewhat surprisingly, this apparently severe constraint still allows an exponential growth in the number of active tasks in the system.

(v) A task can only be stolen if it is deeper in the process tree than any task on the receiving processor.

15.2.4 A General Model of Divide and Conquer Algorithms

We have illustrated earlier how a divide and conquer version of factorial can produce potentially large process trees. More generally, we can define a combinator which captures the essence of the divide and conquer principle:

```
DEF D_C (primitive,divide,combine,solve) = f
  WHERE f x = IF primitive x
             THEN solve x
             ELSE combine (map f (divide x))
             FI
```

map is the function which takes a function and a list and applies the function to every element of the list. D_C is a higher-order function which can be specialised by giving it suitable definitions of primitive, divide, combine, and solve. primitive x returns TRUE if x is a primitive problem, solve x solves a primitive problem, divide x splits a non-primitive problem into a list of subproblems, combine x combines a list of sub-results.

We can use D_C to define factorial:

```
DEF fac n = dfac (1,n)
  WHERE dfac = D_C p s d c
        WHERE p (lo,hi) = (lo=hi)
              s (lo,lo) = lo
              d (lo,hi) = ((lo,mid), (mid+1,hi))
                        WHERE mid = (lo+hi) DIV 2
              c (a,b) = a*b
```

Our definition of the D_C combinator does not restrict us to dividing a complex problem into just two simpler subproblems: the degree of splitting could be three or more because divide returns a list of subproblems and combine processes a list of results.

Examining the definition of the D_C combinator, we can see that exploitation of parallelism depends critically on the way in which the subexpression

```
combine (map f (divide x))
```

is evaluated. To create the potential for parallel evaluation, we want the divide function to create a list of subproblems, and the map function to race along this list creating appropriate sub-tasks. On the other hand, if the lists are very large we would like the architecture to govern their growth so as to match available resources.

For the D_C definition of fac, there is no great problem even with a normal order reducer if we use demand forking across strict operators. A demand for the value of combine would force map f (divide x) to create a list of two objects, and the strict operator * would force parallel evaluation of both arguments.

However, more interesting examples of D_C applications such as sorting do not succumb to this trick. There is a fundamental problem in adopting a declarative programming language (which leaves great freedom for the architecture to exploit parallelism) and then expecting it to express the fine details of control.

This deep problem was recognised early in the ZAPP project, and led to two lines of work. One approach (evidenced by the development of the LNET and DyNe formalisms presented briefly below) attempts to integrate process notions with the lambda calculus. The alternative approach developed annotations for the lambda calculus which are capable of expressing the necessary control. The annotation approach is described by Burton (13). The LNET work was introduced by Kennaway and Sleep (14).

For the purpose of early ZAPP experiments, which were oriented towards divide and conquer algorithms, we effectively built into our simulator a specialised version of the D_C combinator. This mapped, in a single reduction step, a complex problem into a set (not list) of subtasks which could be reduced in parallel.

15.2.5 Using a ZAPP

Notionally, a ZAPP user defines his program as a function based on the D_C combinator, and provides the necessary data. The basic steps in applying a ZAPP to perform the computation are

- (a) Pre-broadcast the program to all ZAPP elements.
- (b) Inject the initial problem (specified by the data) into some selected element of ZAPP.
- (c) Extract and report the final result from the selected ZAPP element.

Alternative methods of using a ZAPP are also envisaged: for example, multi-user operation involving several distinct I/O elements appears feasible. This would in principle allow the "busy" phase of one ZAPP job to overlap the start-up and wind-down phases of other ZAPP jobs.

15.2.6 General Behaviour of a Single-user ZAPP

In the single-user mode outlined above, a ZAPP starts operation by loading each element with a copy of the relevant divide and conquer function. The initial problem is then injected into a distinguished ZAPP element.

This element starts executing the process defined by the initial problem. In doing so, it constructs a stack of tasks very much in the manner of a sequential machine. But whereas a conventional machine would record only the minimal amount of information on the stack, a ZAPP element records - for each "divide" step - full details of all tasks which can be executed concurrently.

The immediate neighbours of the initially active element very soon become aware of the fact that they are idle whilst there is work to be done. These idle neighbours will steal tasks from the stack of the initially active element, and each neighbour will in turn build up its own stack of tasks.

Each task offloaded from one ZAPP element onto another cannot be moved again. If the task moved is primitive it can be executed and the result returned to its logical parent in the process tree. Because of the single steal rule, this must be on an immediate physical neighbour. If on the other hand the task offloaded is not primitive, it must be decomposable into a set of new tasks, which are offloadable. If we assume that every active task creates two sons, and that one is immediately dealt with by the processing element concerned while the other is stolen by a neighbour, then the number of active tasks will initially grow exponentially with time, filling the ZAPP network.

Thus the apparently highly restrictive rule (iv) for offloading and the depth-first rule for individual processing elements in fact allow enormous freedom for a huge process tree to diffuse rapidly. Having ensured that work can diffuse quickly over the network, we must now ensure that the machine does not become overloaded with suspended tasks waiting for the results of other tasks. The basic governing mechanism in a ZAPP is a heuristic for deciding when to steal a task from a neighbour. At one extreme, no offloading at all would result in the entire process tree being handled by one ZAPP element, in the manner of a sequential processor. At the other extreme, offloading at every opportunity could rapidly saturate the individual memories of each element. In practice, the simple expedient of governing growth by restricting the

number of stolen tasks on each element to a small constant appears effective.

In the next section, we describe some early ZAPP simulation results. Broadly, in all these experiments we observed the following phases in running a problem on a ZAPP.

A. The infection phase. Initially, just one ZAPP element is active. The number of active element then rises rapidly (at a rate which is initially exponential if the interconnection topology is rich enough).

B. The busy phase. Roughly, the infection phase splits a large problem into $O(P)$ subproblems, and distributes these to the P ZAPP elements. Ideally, the $O(P)$ subproblems would be of similar size, and - for really large problems - allow a very long busy phase requiring no communication between ZAPP elements. In practice, the split cannot be perfectly balanced, but this is remedied dynamically by the basic ZAPP diffusion mechanism. During the busy phase, every ZAPP element is engaged in productive work.

C. The combination phase. In the ideal case, P subresults are produced simultaneously and rapidly combine via the logical process tree to produce the final result at the initially active element. In practice, the combination phase may be long: it accounts for the major loss of processor utilisation for small problems.

The major result from early ZAPP simulations is that given a large enough process tree with respect to a given ZAPP, the busy phase overwhelms the infection and combination phases, yielding processor utilisations approaching 100%.

15.2.7 Experiments

15.2.7.1 Basic results. A very simple (but very inefficient!) means of computing

$$f k = 2^{k-1}$$

is to use the rule

```
DEF f k = IF k=1 THEN 1
           ELSE (f (k-1)) + 1 + (f (k-1))
```

This can be defined using the D_C combinator:

```
DEF f = D C(p,s,d,c)
        WHERE p k = k=1
              s 1 = 1
              d k = ((k-1), (k-1))
              c (x,y) = 1+x+y
```

It grows a nicely balanced process tree in which each subtask is specified by a single integer. The combining operation is simple addition, so quite large process trees can be run without hitting overflow problems.

Clearly a given k grows a process tree involving $O(2^k)$ elements, so that incrementing k doubles the total work. Fig.15.2 shows a graph of processor utilisation (PU) against k for a 24-element ZAPP which adopted the r - n -cube scheme for interconnection. It can be seen that as the problem size increases, PU rises to approach 100%.

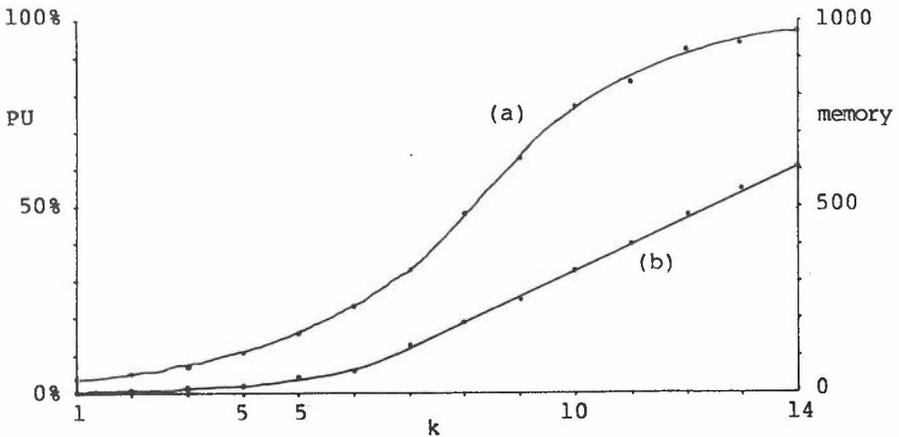


Figure 15.2. (a) Processor utilisation
(b) Memory requirements

The figure also shows the system-wide memory requirements against problem size. After an initially rapid rise, the growth is essentially linear in the depth of the process tree.

The PU figures confirm that as we increase the problem size, the busy phase of a ZAPP run increasingly dominates the infection and combination phases. Because each ZAPP element adopts a depth-first exploration of the process tree, its memory requirements will be related to tree depth just as with a conventional sequential machine. The memory figures suggest that linear growth with tree depth still holds when we introduce parallelism by allowing elements to offload tasks to neighbours under ZAPP rules.

The figures for memory requirements are the "high tide marks" of total memory used by all processing elements. We have also studied the maximum memory required by any one processor, and have found that this also grows linearly with tree depth. We can see that this is ensured by rule (v) of

section 2.3 allowing a processor having suspended tasks but no active tasks to steal from its neighbours only active tasks which are deeper in the process tree than any of its own tasks.

These experiments assumed perfectly balanced process trees. For unbalanced trees, the relevant datum determining memory requirements is the maximum depth of the tree. A perfectly balanced tree with N nodes has $O(\log N)$ depth. A random tree has depth, on average, $O(\sqrt{N})$. A totally unbalanced tree has depth $O(N)$. This implies that it is important to design one's algorithms so as to generate reasonably balanced trees.

15.2.7.2 Further experiments. We simulated a variety of physical networks with up to 2000 elements and various interconnection topologies. The basic ZAPP results (approaching 100% PU for large problems, linear memory requirements for balanced process trees) were confirmed.

Although our early experiments concentrated on the r - n -cube topology (11) we also ran experiments on other topologies. The more advanced topologies such as the r - n -cube and the cube-connected cycles (Preparata (15)) produced better results, but the basic ZAPP results emerged even from simple grids. We concluded that the main effect of more advanced topologies was to shorten the infection and combination phases. For large problems these phases are a minor part, and even a ring could yield good PU.

Following (12) and others we conjecture that the choice of topology should be an engineering compromise between ideal high connectivity and the economic limits of state of the art technology.

The next advance in technology will be wafer scale integration (WSI). For economic reasons, it should be possible to use wafers which have a proportion of inoperative processing elements. We ran some studies of a ZAPP in which a randomly selected proportion of elements were considered inoperable. So long as the remaining elements formed a connected graph the basic ZAPP results continued to hold.

15.3 THEORETICAL RESULTS

15.3.1 Rewrite Systems

A powerful method for implementing functional languages (and other languages, for that matter) is by the use of rewrite rules. A rewrite rule has a left hand side and a right hand side and states that any expression having the form of the left hand side may be replaced by the right hand side. An expression is "evaluated" by applying the rewrite rules as often as possible, eventually arriving at an expression to which no rule applies. Such an expression is

called a normal form and is the "value" of the original expression.

The classical example of a language defined by rewrite rules is the lambda calculus, with the rule of beta reduction: $(\lambda x.E)F \rightarrow E[F/x]$. This rule states that an application of a lambda abstraction to any other expression can be replaced by the result of substituting that expression for every occurrence of the bound variable of the abstraction within its body. (We ignore, for simplicity, technicalities concerning free and bound variables and alpha conversion.) Another example is SASL (7), which is a functional language in which the programmer writes his own rewrite rules.

In the description above of beta reduction, we spoke as if the expression being reduced was represented as a string, or perhaps a parse tree. For practical implementation, however, string rewriting has a major disadvantage. A rewrite rule may demand that some part of the expression it is applied to be copied many times. Beta reduction of $(\lambda x.E)F$, for example, will make a new copy of F for every free occurrence of x in E . This not only takes up space, but if F itself contains redexes, then each of its new copies will need to be evaluated separately.

This problem can be avoided if we interpret the rewrite rules as applying, not to strings or parse-trees, but to graphs. Where a tree may have multiple occurrences of a subexpression, a graph can have just a single copy of that subexpression and many references to it.

15.3.2 Graph Rewriting on ZAPP

Graph rewriting systems should be naturally suited to execution on a ZAPP machine. Each node of the graph is represented as a process, and each edge of the graph corresponds to a possible line of communication between processes. The network of processes is wrapped around the physical network of processors, and the behaviours of the processes are designed so that the action of the entire network mimics the application of the rewrite rules.

15.3.2.1 Fine grained beta reduction. A direct implementation of beta reduction for lambda expressions would have the problem that a single beta reduction can involve an arbitrarily large part of the whole graph, which may be spread over a large part of the physical network. Exclusive access to the whole of a beta redex while it is being reduced would severely limit the parallelism available. Beta reduction is too coarse a unit of computation. The rewrite systems best suited for highly parallel execution are those whose redexes are small - no more than a few nodes. We have developed such a "fine-grained" rewrite system for lambda calculus which is equivalent to beta reduction, in the sense that for any

lambda expression which has a normal form with respect to beta reduction, fine-grained reduction will produce the same normal form, and vice versa.

15.3.2.2 Combinators and directors. Combinators provide another graphical implementation for functional languages (Turner (16,17)). Any lambda-expression can be translated into a combinator graph, which can then be evaluated by applying simple rewrite rules to small portions of the graph. However, there is a performance problem with the translation into combinators. We have shown (18) that in the worst case a lambda expression of size n may turn into a combinator graph of size $O(n^2)$. The problem lies with expressions whose parse-trees are highly unbalanced, having a height proportional to the number of their leaves, and with deeply nested lambda-abstractions.

One remedy for this is to preprocess lambda expressions so as to turn them into equivalent, but less unbalanced, expressions. Suppose we have a highly unbalanced expression E . We select some node of E about halfway from the root of E to its deepest leaf. Let F be the subexpression whose root is this node. We invent a new variable name x , and replace E by the expression $(\lambda x.E[x/F])F$, where $E[x/F]$ denotes the result of substituting x for F in E . This replaces a tall, thin tree by a shorter, fatter one. For the transformation to be valid, F must not contain any free variable which is bound by any lambda-node on the path from the root of E to the root of F . The transformation is then recursively applied to both $E[x/F]$ and F . Burton (19) shows that for a restricted class of lambda expressions, a suitable node can always be found, and the whole transformation can be performed in linear time. The resulting expression can then be translated into combinators in linear time with only linear expansion. This method is not applicable to all lambda-expressions, however. Expressions must, roughly speaking, not contain functions with global variables. Although every lambda expression can be transformed into an equivalent one with this property (by supplying each function with its global variables as extra parameters), that transformation may introduce a quadratic expansion in the worst case.

A second method of overcoming the quadratic complexity uses a simplification of combinators, which we call directors. Firstly, we regard the combinators introduced by the translation as being attached as annotations to the interior nodes of the parse-tree, rather than as ordinary constants appearing at the leaves. It then becomes clear that the purpose of each combinator is to direct an incoming argument to one or other, both, or neither of the two descendants of the apply node it is attached to. We therefore replace the S and S' combinators, both of which direct an argument both ways, by the symbol \wedge . Similarly, we replace B and B' by \backslash (send right), and C and C' by $/$ (send left). K is replaced by $\#$, and I is retained. The

We are therefore obliged to develop a language for the purpose. A secondary goal of our design is to give the language as functional a style as possible. A node of a graph expression should be modelled by a process which may receive a request for the value of the expression, and reply with that value, possibly having engaged in other communications in order to compute it. The communication primitive of the language is therefore a two-part exchange of messages in which one process sends a message to another and receives a reply.

Our first design was called LNET. Its principal features are the following.

1. A process in LNET consists of two components: a name, by which it is addressed by other processes, and a script, which specifies its behaviour.

2. The two partners to a communication adopt asymmetric roles, called active and passive. The syntactic representations of active and passive communications are $e?n!t$ and $t?!e$. Here e ranges over expressions, n over process names, and t over templates. A template is used for pattern matching, and is simply an expression in which certain occurrences of variables are marked as being binding occurrences. When one process attempts the active communication $e!n?t$ it is suspended until the process named n attempts a passive communication $t?!e$ in which t matches e . When this happens, e is evaluated, the bound variables of t are bound to the corresponding components of the value of e , and then e is evaluated and matched to t in the same way. After the communication is completed, the processes go their separate ways. The points to notice about this form of communication are:

(a) Pattern matching is built into LNET communication.

(b) The active partner is committed to completing the communication. If the passive partner never accepts the active offer, then the active partner will be deadlocked for ever.

(c) The active partner directs its offer at a particular process; the passive partner is willing to accept any active offer that matches its template.

(d) The exchange of messages appears to both partners as an indivisible action, although it may be implemented as two separate message-passings.

(e) Scripts can be sent as messages from one process to another.

3. A behaviour p can consist of an alternative choice between two or more behaviours. If each of the components begins with a passive communication, then the choice will

depend on which of the templates of those communications is first matched by an active offer. If, on the other hand, one or more of the components begins with an active communication, then that components may be chosen unilaterally. The difference is between a choice which may be controlled from outside a process and one that cannot be.

4. A process may create processes running in parallel with it.

5. Processes may be parameterised.

6. A process may execute the primitive action wait, which has the effect that the process is suspended until some other process makes it an active offer. This is a rather ad hoc addition to the language which we later managed to remove.

7. A process may be an indirection process which merely relays active offers to some other process. This is another ad hoc feature which was later removed.

LNET is able to express combinator graph reduction and director string reduction. It has been given a formal axiomatic semantics (14). However, LNET is overcomplicated. In particular, wait and indirection processes are somewhat arbitrary creations. In modelling graph reduction, the purpose of wait is to allow some control over which parts of the graph are allowed to be reduced, and in what order. The maximally parallel algorithm which evaluates everything in sight is not necessarily the best. Even a machine with a high degree of actual parallelism may be flooded by this algorithm with computations whose results will never be used (for example, because they are on the unselected arm of a conditional). Indirection processes play much the same role as indirection nodes in conventional graph reducers (e.g. Wadsworth (27)).

We have developed a successor to LNET which removes these flaws. DyNe (Dynamic Networks) differs from LNET principally in allowing the passive partner in a communication to engage in other computation and in other communications between accepting an active offer and making its reply.

In DyNe, as in LNET, a process consists of a name and a script. A script can be a sequential composition of scripts, an alternative composition, a parallel composition, a basic action, or a basic constant. An action is either a passive communication, an active communication, or the creation of a set of parallel processes. Variables can also be defined by declarations. The formal syntax is, in outline, as follows.

```
S ::= S . S           (sequential composition)
     S + S           (alternative composition)
```

```

S | ... | S    (parallel composition/tupling)
A              (action)
B              (built-in script)

A ::= GET T DO S REPLY S    (passive communication)
S -> S ? T                (active communication)
CREATE I:S | ... | I:S ENDCREATE
                          (process creation)

```

There is no separate category of expressions. Basic constants and the built-in operations on them are themselves scripts, the built-in scripts ranged over by B above. This class includes integers, booleans, tokens (a countable set on which there is only an equality operator), and some standard repertoire of arithmetic and logical operators. Considered as behaviours, the basic constants all specify the empty behaviour of a terminated process. These scripts are the final result of executing a script. The built-in operators, on the other hand, appear to the processes using them like scripts, which repeatedly accept an active offer of a tuple of values appropriate to the operator and reply with the result of applying the operator to the values. There are also declarations of the conventional sort which bind identifiers to scripts.

T ranges over templates. A template is either a basic constant, a tuple of templates, or of the form I:t. I is an identifier and t is a type. Types are formed from the set INT, BOOL, TOKEN, NAME by tupling, union, and difference. The matching operation between templates and expressions should be sufficiently obvious to need no further explanation.

Process names do not appear in the syntax. When a process executes the action `CREATE I1:S1 | ... | Ik:Sk ENDCREATE`, which creates k new processes, k new process names are generated and bound to the identifiers I1..Ik. Process names can be sent as messages, compared for equality, and used for pattern-matching.

We have an axiomatic semantics for DyNe and are currently working on a denotational semantics. This will involve a generalisation of the techniques of Kennaway (28) to handle the unique feature of DyNe that scripts can themselves be sent as messages.

15.4 CONCLUSIONS

The simulation results show that a simple set of local rules allows a large process tree to diffuse across a richly connected set of processing elements. We have shown that depth-first priority schemes potentially allow exponential growth of the initial problem over the network. We also showed how the depth-first scheme controls memory demands. It is interesting to note that Watson (29) independently found that breadth-first schemes can lead to unacceptable

memory overheads, and remedied the problem by converting a hardware queue in the Manchester dataflow machine into a stack (which naturally imposes a depth-first ordering).

Early ZAPP results confirmed our intuition that it is possible to "buy speed" from a simple diffusion mechanism. Although we initially considered only simple divide and conquer algorithms, later ZAPP work (3) confirmed early results for more interesting examples using a programmable version of the simulator.

Our theory work identified many important problems. We characterised the space problem for Turner combinators as quadratic (18), and offered two solutions (19,20). We developed fine-grain (combinator-like) schemes for evaluating functional languages. We also developed a range of safe reduction orders, which naturally includes the usual normal order. Perhaps most importantly, we recognised the deep problem of naturally integrating control information into functional languages, and proposed two distinct approaches.

15.5 ACKNOWLEDGEMENTS

Warren Burton designed and programmed the first ZAPP simulator (called DREAM) which we used to obtain the basic results. M.M.Huntbach ran many of the later experiments and considerably extended DREAM by making it programmable. We take great pleasure in acknowledging their contributions to ZAPP.

The work on ZAPP was supported by the SERC Distributed Computing Systems programme under grants GR/B/29221, GR/B/62297, and GR/C/35402. Particular thanks are due to DCS coordinators Hopgood, Witty, and Duce for encouraging us and also for arranging many fruitful workshops.

REFERENCES

1. Kung, H.T., 1979, "Let's design algorithms for VLSI", Report CMU-CS-79-151, Carnegie-Mellon University.
2. Kennaway, J.R. and Sleep, M.R., 1983, "Novel architectures for declarative languages", Software and Microsystems, 2, 59-70.
3. Burton, F.W. and Huntbach, M.M., 1984, "Virtual tree machines", IEEE Trans. on Computers, C-33, 278-280.
4. Barendregt, H., 1980, "The lambda calculus", North-Holland.
5. Landin, P., 1966, "The next 700 programming languages", Comm.ACM, 9, 157-166.
6. Gordon, M.J., Milner, R., and Wadsworth, C.P., 1979,

- "Edinburgh LCF", Lecture Notes in Computer Science, vol.78. Springer-Verlag.
7. Turner, D., "SASL language manual", University of St.Andrews.
 8. Burstall, R., MacQueen, D.B., and Sannella, D.T., 1980, "HOPE: An experimental applicative language", Report CSR-62-80, Department of Computer Science, University of Edinburgh.
 9. Henderson, P., Jones, G.A., and Jones, S.B., 1983, "The Lispkit Manual", Oxford University Computing Laboratory Programming Research Group Report PRG-32.
 10. Kowalski, R., 1979, "Algorithm = Logic + Control", Comm.ACM, 22, 424-436.
 11. Burton, F.W. and Sleep, M.R., 1981, "Executing functional programs on a virtual tree of processors", Proc. ACM Conf. on LISP and Functional Programming, Portsmouth, New Hampshire.
 12. Hillis, W.D., 1981, "The connection machine", AI Memo No.646, MIT Artificial Intelligence Laboratory.
 13. Burton, F.W., 1984, "Annotations to control parallelism and reduction order in the distributed evaluation of functional programs", ACM Trans. on Programming Languages and Systems, 6, 159-174.
 14. Kennaway, J.R. and Sleep, M.R., 1983, "LNET: syntax and semantics of a parallel language", University of East Anglia.
 15. Preparata, F.P. and Vuillemin, J., 1979, "The cube connected cycles: a versatile network for parallel computation", Proc. 20th IEEE Conf. on Foundations of Computer Science.
 16. Turner, D., 1979, "A new implementation technique for applicative languages", Software: Practice and Experience, 9, 31-49.
 17. Turner, D., 1979, "Another algorithm for bracket abstraction", J. Symbolic Logic, 44, 267-270.
 18. Kennaway, J.R., 1982, "The complexity of a translation of lambda calculus to combinators", Report CS/82/023/E, University of East Anglia.
 19. Burton, F.W., 1982, "A linear space translation of functional programs to Turner combinators", Inf.Proc.Letters, 14, 201-204.
 20. Kennaway, J.R. and Sleep, M.R., 1983, "Counting

- director strings", Report DSAG-1, University of East Anglia.
21. Dijkstra, E.W., 1980, "A mild variant of combinatory logic", Report EWD735.
 22. Noshita, K., 1984, "Translation of Turner combinators in $O(n \log n)$ space", University of Electro-communications, Tokyo.
 23. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983, 1983, "The Programming Language Ada Reference Manual" Lecture Notes in Computer Science, vol.155, Springer-Verlag.
 24. Milner, R., 1980, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, vol.92, Springer-Verlag.
 25. Hoare, C.A.R., 1978, "Communicating Sequential Processes", Comm.ACM, 21, 666-677.
 26. Inmos, Ltd. 1984, "occam Programming Manual", Prentice-Hall.
 27. Wadsworth, C., 1967, "Semantics and Pragmatics of the lambda calculus", D.Phil. thesis, Oxford.
 28. Kennaway, J.R., 1981, "Formal semantics of parallelism and nondeterminism", D.Phil. thesis, University of Oxford.
 29. Watson, I. Private communication.

Chapter 16

The Manchester dataflow project

J. R. Gurd, I. Watson, C. Kirkham

16.1 INTRODUCTION

Simple block diagrams of novel computer architectures can belie complex implementation issues which need resolving before the intended system function is realised. This fact of engineering life has been entertainingly documented for a well-known series of supercomputer by Lincoln (1). He notes that nontechnical matters can influence the implementation of systems just as profoundly as technical issues. The paper below describes some of the technical problems encountered by the Manchester Dataflow Research Group during implementation of their prototype dataflow computer, and outlines how each difficulty was resolved, or bypassed, in the course of the project. As far as possible only those issues which are peculiar to the dataflow approach are addressed, but seasoned system implementers will recognise that many of the themes are quite general. Software and evaluation matters are covered, as well as details of engineering construction.

16.1.1 Brief History of the Manchester Project

Work on dataflow computation at Manchester was initiated in late 1975 and early 1976 by Gurd, Treleaven and Watson. Preliminary dataflow machine architectures were designed on paper and compared by gedanken-experiment, as described by Gurd and Treleaven (2) and in Treleaven's Ph.D. Thesis (3). After this, work continued on the development of a general purpose 'labelled-token' dataflow model of computation and a machine for its execution, as described by Gurd et al (4). A related project studied a more specialised dataflow architecture for real-time control of a robot arm, as described by Egan (5).

The general purpose architecture was developed via extensive simulation experiments using a Pascal program to interpret the evolving ordercode. Construction of a prototype version of the machine commenced in late 1978 under SERC/DCS funding. As described in section 16.2.1, the machine structure was altered slightly during the construction period. The prototype hardware comprised a 36kbyte instruction store, a 240kbyte data store, and six microcoded function units. It executed its first program, a doubly-recursive algorithm for the factorial function

written in the experimental high-level programming language Mad, on the 6th October 1981. The first truly parallel program execution took place on 9th November 1981.

The prototype system was designed and constructed by a team of just three engineers in three years. It contained some 4000 MSI integrated circuits and was front-ended by an LSI11/03 microprocessor. Since the prototype became operational, work has progressed on five main fronts.

Firstly the hardware configuration has been expanded to include 288kbytes of instruction store, 15Mbytes of data store and 20 microcoded function units. This has increased the number of integrated circuits in the system to nearly 10000. The front-end processor has been upgraded to a VAX11/780.

Secondly a hierarchy of standard dataflow programming language interfaces have been designed and implemented. The levels of the hierarchy roughly correspond to the levels of conventional language systems. At the lowest level there are disassembler and assembler programs which convert binary machine commands to and from a textual format with one line of text per machine word. Above this there are template (macro-level) assemblers which translate the intermediate target languages used by the high-level compilers. Software simulators for the dataflow hardware have also been written, providing various degrees of accuracy in predicting the hardware behaviour.

Thirdly a number of benchmark programs have been developed, written in a variety of parallel programming languages. Small versions of these programs have been used as the basis for a preliminary performance evaluation of the single-ring hardware system. Results for such programs (up to 4k instructions and 10k tokens of data) show that the internal parallelism of the ring hardware (about 30-fold) can be exploited effectively by programs with a comparable degree of parallelism. Work is now in progress to expand the scope of the evaluation by executing larger programs which perform more realistic computations. Programs with 32k instructions and using up to 100k tokens are currently being tested.

Fourthly a small-scale hardware simulator of a multi-ring dataflow system has been constructed to evaluate multiprocessor task distribution strategies. Each processor in the simulator comprises a 256kbyte code/data store plus two 68000 microprocessors which simulate the modules of the large-scale ring. Each processor uses 200 MSI integrated circuits, in addition to the 68000s. One of these processors is roughly one thousandth as powerful as a full-size ring. The current simulator configuration comprises four processors connected together by a 4x4 switch module and containing nearly 900 integrated circuits. A sixteen processor version containing 3500 integrated circuits is nearing completion.

Finally, theoretical work has been in progress to establish a semantic framework for describing dataflow program transformations which might be used for code improvement during compilation of highly-parallel programs.

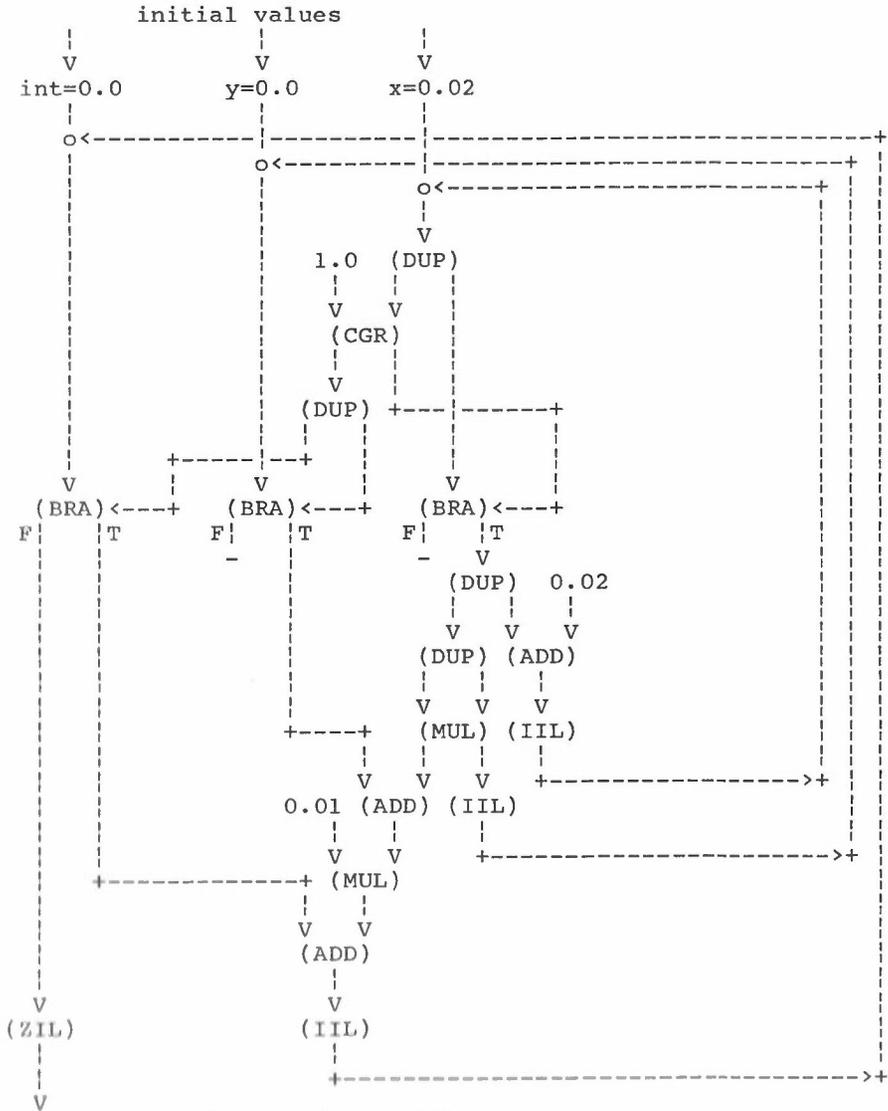
16.1.2 Tagged-Token Dataflow Computation.

The model of computation implemented by the Manchester hardware is known more generally as tagged-token dataflow. It was discovered independently by Arvind et al (6), and owes much in concept to the pioneering work of Dennis (7).

Dataflow programs are directed graphs in which the nodes represent instructions (which can be regarded as pure operators) and the arcs represent inter-instruction data paths. It is important to realise that the arcs do not behave as first-in-first-out queues. Data is transmitted along the arcs in tagged packets known as tokens. Tokens in the Manchester machine carry three independent tag-fields known as the activation name, the iteration level and the index. These are used to separate independent activations of, respectively, functions, loops-in-time and loops-in-space, as described by Gurd and Watson (8). Tags are used in loops to simulate first-in-first-out queues on arcs. Such queues are inappropriate for recursive functions which need tags to generate a parallel environment analogous to the 'stack' environment used in sequential computers.

The sample program (Fig.16.1) shows the Manchester machine-code for an iterative loop which computes the area under the curve $y=x^2$ between $x=0$ and $x=1$ using a trapezoidal approximation with constant x intervals of 0.02 units. Note the use of tag-manipulating instructions (increment iteration level, zero iteration level) to ensure correct queueing of the loop termination control tokens at the inputs to the branch instructions. Note also the use of explicit duplicate instructions which replicate data required at two or more subsequent instructions. The Manchester system imposes a maximum fan-out from each instruction of two, so chains of duplicates must be used for larger fan-out. In some circumstances it is possible for a subsequent duplicate to be incorporated into the preceding instruction (as shown in Fig.16.1). Two is also the maximum possible number of inputs to an instruction, due to the way in which tokens travelling to the same instance of an instruction are matched together. Manchester instructions are thus monadic or dyadic only. Certain monadic instructions are formed by dyadic operators with one fixed (literal) input (as shown in Fig.16.1).

Another important point is the use of branch instructions which act as 'switches' in the arcs of program graphs. These are used to implement conditionals, loops and recursion. Each branch is controlled by a Boolean control input which is shown entering the instruction from the right-hand side. If the value of the token on this input is false, then the other incoming token (on the top input) is sent down the left-hand output (labelled F in Fig.16.1), otherwise the control is true, and the other input token is sent down the right-hand output (labelled T). Note that branch instructions can be used as 'gates' which pass a value or destroy it, according to the Boolean control value, simply by leaving one of the output arcs unused (as shown in Fig.16.1).



key to instructions:

- int ADD - add floating point values
- final BRA - branch (see text)
- value CGR - compare floating point l.h. > r.h.
- DUP - duplicate (see text)
- IIL - increment iteration level (see text)
- MUL - multiply floating point values
- ZIL - zero iteration level (see text)

Fig.16.1 Sample dataflow program graph

16.2 ENGINEERING CONSTRUCTION

By far the largest effort in the early part of the project was expended on hardware engineering tasks. Foremost of these was to freeze the overall system design so that individual engineers could proceed to design their modules independently. It was also imperative to define the interfaces between modules so that no mismatches in design could occur. These proved to be difficult decisions to make, but ones which had great ramifications for the later progress of the project. Equally important, though rather less exciting, was the choice of physical environment components such as boards, cabinets, cables and power supplies. These choices also affected the eventual provision of computer-aided design tools for board production.

Undoubtedly the most important individual module design was that of the token-matching module. This required an associative style of memory which was too expensive to implement directly and so had to be simulated using hashing techniques. The module has been rebuilt several times during the project, and experiments to discover the optimum hashing function are still in progress. More recently it has become apparent that large programs will require extensions to the originally envisaged architectural structure. It is planned to implement a specialised data structure store module, and this will affect the way the multi-ring switch module will operate.

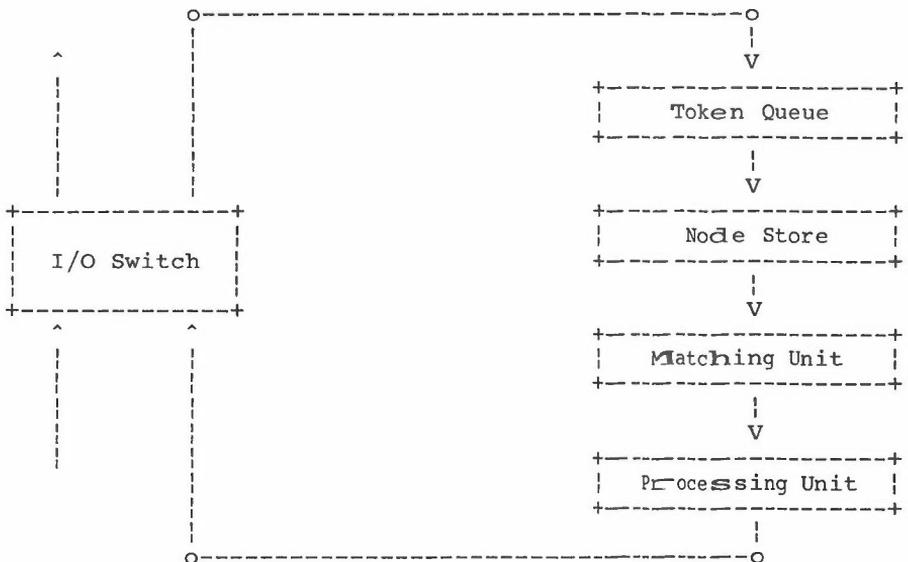


Fig.16.2 Initial proposed dataflow system structure

16.2.1 System Architecture

It was quickly decided that the system would be ring-structured and would contain four major modules, known as the Token Queue, the Node Store, the Matching Unit and the Processing Unit, all connected to a host system via a 2x2 Switch Unit. At first it was decided that the modules should be linked together in the order given above (Fig.16.2). Functional descriptions of the major modules were defined, and a software simulator for the system was written. Subsequent use of the simulator, particularly for implementation of the Lapse language by Glauert (9), generated additional requirements for the instruction-set and other module functions. It also elicited certain important program characteristics, such as the frequency of occurrence of monadic and dyadic instructions, which determines the rate of production of matched token-pairs from the Matching Unit.

Further hardware design studies determined the clock characteristics for the system and the inter-module interface (see section 16.2.3). This work also revealed that an unnecessarily high bandwidth was required of the Node Store. By rearranging the order of the Node Store and Matching Unit (Fig.16.3), the required store access time was increased from 200 to 300 nanoseconds (due to the expected rate of token-matches). The differences between this structure and the original proposal can be seen by examining the 1st and 3rd versions of the report by Gurd et al (4).

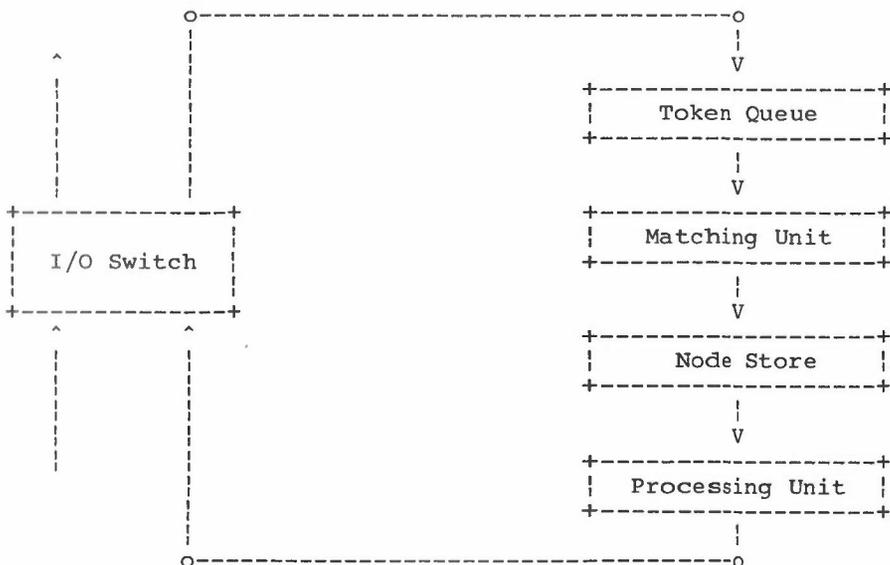


Fig.16.3 Eventual dataflow system structure

The most important ramification of this reorganisation of the system was the necessity to specify a matching function on each data token entering the Matching Unit. The matching function determines whether the instruction to be executed is monadic or dyadic. Tokens to the former can bypass the matching process, whereas tokens to the latter must match up with their partner. More exotic matching functions were also introduced to cope with some of the programming practices discussed below.

16.2.2 Physical Construction

Priorities in defining the physical construction of an electronic system are always difficult to ascribe. Most of the required decisions are made arbitrarily, but they have a cumulative effect which often overshadows the seemingly more germane decisions governing the abstract system architecture. In our case the dominant design decision was to use printed circuit boards (PCBs) with a fixed size of 14.5 x 12 inches (triple-Eurocards) using three 64- or 96-way connectors per board, and installing the boards in a standard 19-inch housing. We expected to be able to fit between 100 and 200 MSI integrated circuits onto each board. Initial design studies indicated that each module would occupy one 19-inch 9U rack, and that a complete ring could be housed in a 7-foot high cabinet. Subsequent implementation decisions have been coloured by the need to conform to this physical organisation.

At the end, the prototype system comprised 15 different PCB designs, with between 50 and 250 integrated circuits on each board. The most densely populated boards proved significantly difficult to lay out by hand, and caused us to develop automated design tools for subsequent construction work (see section 16.2.4).

16.2.3 The Inter-Module Interface

In designing the interface between modules, attention was paid to the physical size of each module and the scheme for distributing clock signals around the system. It was clear that a multi-ring system could not be driven synchronously (with a single, common clock) because of the large distances which would separate the component rings. It was further decided that clock signals could not be transmitted reliably between racks (modules) and so the decision was taken to implement each module with its own internal clock and to define an asynchronous protocol for transmitting data from one module to another. Module designers were to be left relatively free to choose an appropriate internal clock speed.

The circuit used to implement the asynchronous interface has been published by Gurd and Watson (10). It was designed to operate at clock speeds up to 40MHz at either end, although it later transpired that the decision elements in the 74S112 flip-flops used were unable to support clock rates greater than 30M Hz (the design

incorrectly assumed that the JK-type 74S112 used the same decision circuit as the D-type 74S74). This speed loss was discovered after construction of the Token Queue module, and subsequent designs were adjusted to use slower module clocks without significantly affecting the performance of the system.

16.2.4 CAD Tools

The chosen physical construction constituted a change in practice from previous projects. Consequently there were no design automation tools available to help lay out the first boards. The most complex of these was the Matching Unit store board which contained over 200 integrated circuits. This board, and the equally complex function unit board for the Processing Unit, took three months elapsed time each to track by hand. This delay was considered inordinate, and forced us to develop some layout tools for design of PCBs for later enhancement of the system.

The first tools have not circumvented the need for hand layout of the boards, but they allow rapid transfer of the layout to film via automatic digitising and graphical editing. Some two thirds of the board design time has been saved by these aids alone. Work is now in progress to produce automated placement and tracking systems which are capable of producing a board directly from its logic diagram. Of course, such aids have long been a part of the industrial designer's toolkit, but they are expensive, and the Universities are mostly devoid of such assistance. In all, we estimate that we spent nearly a quarter of the engineering effort during the project on developing computer aided design tools for PCB production.

16.2.5 The Matching Unit

The Matching Unit is the critical module in any dataflow processor. Everything else can be pipelined or made parallel according to the data rates expected from the matching process. The act of matching together tokens 'drives' the rest of the system into action.

Matching is essentially an associative process. For the simplest dyadic matching function, tokens arrive at the Matching Unit looking for their partners which are held nearby in a 'pool' of unmatched tokens. Each token searches the pool associatively for its partner. If the partner is present in the pool it is extracted and attached to the incoming token to form an operand pair which can find and execute its destination instruction. Otherwise the incoming token must be the first to arrive, and it is merely placed in the pool to await its partner.

Unfortunately, associative storage is prohibitively expensive and so the unmatched token pool has to be simulated using conventional random-access memory. The structure of this 'pseudo-associative' memory is described by da Silva and Watson (11). It is based on a parallel hashing scheme whose performance depends critically on the

efficiency of the hash function. When uneven distribution of tokens occurs, because of an inappropriate hash function, the pseudo-associative store overflows causing tokens to be sent to a parallel Overflow Unit. This simulates the associative token pool by semi-sequential search through a random-access store. This is a slow process which must be kept to a minimum. However, selection of a suitable hash function has proved to be another unexpectedly difficult task which has not yet been completed satisfactorily.

The picture is further complicated when we consider the more complex matching functions. These may cause data to reside in the unmatched token pool even after matching has occurred. They may also cause generation of defer tokens which circulate around the processor ring without performing useful work. The problems of clearing up garbage from the unmatched token pool and of minimising the number of defer tokens are substantial. They are still receiving concerted attention.

16.2.6 The Structure Store

Recent studies of large benchmark programs have indicated that the Matching Unit store can become dangerously overloaded with long-residence tokens which are accessed in the manner of tables or databases. This suggests that data storage should be separated into two parts, for short-term and long-term storage, respectively. The Matching Unit is well-suited to short-term data, but the natural counterpart for long-term data is not obvious.

We are currently investigating a Structure Store unit for long-term data storage. The scheme was first recommended at Manchester by Bowen (12), based on work at MIT by Misunas (13) and Arvind and Thomas (14). The scheme is based on the assumption that long-term data is usually structured in such a way that large amounts of data are logically grouped together and can be referenced by a single 'pointer' or 'descriptor' token, whilst being stored in a remote storage area. The advantage of such a scheme is that the elements of each structure can be stored without the tag fields which accompany unstructured tokens around the ring. This leads to a substantial saving in storage space, and also reduces instruction execution time when structures are passed across tag-changing boundaries (eg at the entrance or exit of loops or functions).

The Structure Store will be connected to the prototype ring by a second 2x2 switch module inserted between the Processing Unit and the I/O Switch. The new unit can be viewed as a second ring which has a specialised instruction-set suitable for handling stored structures.

16.2.7 Multi-Ring Systems

For the future there is considerable interest in dataflow multiprocessors in which the basic single-ring processor is replicated and an interconnection network is introduced to link many of them together. There would also

be the possibility of replicating Structure Store rings. Such systems would be capable of very high rates of computing, with the chance of incrementally increasing the rate by adding extra rings to the system.

The problem to be faced in realising such a system is that of distributing the available work evenly across a potentially huge processing space. This is a challenge to any kind of multiprocessor and one which has barely been addressed in any current design. We have commenced simulation studies in hardware and software by which we hope to demonstrate the feasibility of large-scale dataflow multiprocessors in the near future.

16.3 LANGUAGE SYSTEMS

The strategy for designing language systems for the dataflow machine has been one of trial and error. We have tried a number of experiments with different styles of high-level language and come to some preliminary conclusions about the feasibility of their implementation.

At first there was pressure to implement compilers for conventional languages such as Fortran and Pascal. This did not look promising given the inherently sequential semantics of these languages. Nevertheless, two such compilers have been implemented.

A more promising approach was discovered in the 'single-assignment' languages, whose semantics are nonsequential, and which can be easily translated into directed graph form. This approach has subsequently been adopted by every dataflow research group as the most likely route to efficient code generation. There are indications that these languages are suitable for other forms of multiprocessor besides dataflow systems.

A third approach is to compile from nonprocedural (declarative, applicative, functional, logic) languages into dataflow machine code. Pure functional languages look particularly attractive for this because dataflow instruction execution has distinctly functional semantics. Several experiments have been performed in this area.

A fourth area which has been studied is the implementation of high-level nondeterministic programming languages for handling 'real-time' applications such as operating systems and databases.

Finally, we should mention that compilation from these languages has not been directly to dataflow object code, but has invariably used some form of intermediate code. The similarity between intermediate codes for all the languages used led us to develop two standard compiler target languages which have been used for more recent compilers.

16.3.1 P0 and Pascal

Conventional languages for the Manchester dataflow system have been studied by Whitelock (15) and Veen (16). These studies indicate that it is feasible to translate most conventional language constructs into dataflow code.

However, it is difficult to ensure that all the available parallelism has been extracted, especially for languages which permit aliasing of variables (which in practice means all useful conventional languages). The work of Kuck et al (17) demonstrates that inroads can be made into this problem by recognising special cases, but it remains hard work and cannot cope with all possible situations.

16.3.2 Lapse and Mad

Early efforts at Manchester to 'break the von Neumann mould' of languages centred on two experimental single-assignment languages, Lapse (designed by Glauert (9)) and Mad (designed by Bowen (12)). It is no coincidence that these languages both resemble Id (designed at UC Irvine by Arvind et al (6)). Restriction of assignment so that variables may be assigned values at only one point in the program liberates the single-assignment languages from many of the side-effects that plague conventional programming languages. This liberation is useful in more than one respect. Firstly it dispenses with the need to execute statements in sequential order, as noted by Chamberlin (18). Secondly it makes it easier to prove properties of programs, as demonstrated by Ashcroft and Wadge (19). Thirdly it makes the languages easier to teach, especially to computer novices, as noticed by Tesler and Enea (20).

From a dataflow point of view, the chief advantage of single-assignment languages is that they are easy to compile into dataflow object code. They can be thought of as direct high-level textual representations of directed graphs. They can also be thought of as nonprocedural languages with first order functional semantics.

At Manchester, the Lapse compiler was used in conjunction with the original system structure (see section 16.2.1) and was instrumental in causing changes to the instruction-set which were incorporated into the new system structure. The Mad compiler evolved around the new architecture and has been instrumental in demonstrating the options for implementing data structures in dataflow. In particular, complex matching functions were used to implement stored stream data structures.

16.3.3 Lucid and SASL

The single-assignment languages have been used by the dataflow community in an inherently procedural way. Since they have much in common with nonprocedural functional languages it is natural to attempt implementation of the latter. This has been done at Manchester for the languages Lucid (Ashcroft and Wadge (19), see Bush (21) and Sargeant (22)) and SASL (Turner (23), see Richmond (24)).

This work revealed two major problems. Firstly it transpired that the fully eager dataflow evaluation ordering was not the most efficient way of computing, although it exhibited substantial amounts of parallelism. Some measure of demand drive is needed to control the growth of activity

in a program execution. This is not difficult to achieve in principle, but demand driven access to data structures led to the design of the specialised Structure Store hardware because of the inefficiency of its implementation based on the Matching Unit. Secondly implementation of fully general higher order functions turns out to be contorted. This language feature may be rather more general than required for widespread use. In practice it appears that a small set of predefined higher order functions may suffice, and these could be implemented straightforwardly in dataflow.

16.3.4 DP, CSP and Id Resource Managers

Implementation of high-level nondeterministic programming languages was studied by Catto (25). This work demonstrates how the basic nondeterministic features of the languages DP (Distributed Processes), CSP (Communicating Sequential Processes) and Id can be executed via Manchester dataflow graphs. The work places particular emphasis on the provision of machine level features, such as matching functions, which are essential to efficient handling of nondeterminism. The study was performed in abstract and no compiler was produced. Informal proof techniques were developed, and these have been subsequently developed to form the basis of a theory for reasoning about dataflow programs.

16.3.5 Assembly-Level Languages

When the first user programs were written they tended to be small and could be developed in graph form. To help with translation to machine-code, we developed a macroassembler (ASSEM) which allowed users to define their own 'nodes' and interconnect them with directed arcs. At a later stage, we noticed that a common factor of all the early language implementation work was the use of intermediate compiler target languages based on graphs. However, these languages were dissimilar to the macroassembler language, relying on the concept of templates in which generic nodes are connected together by generic arcs. This observation caused us to develop a Template Assembler (TASS) which is now used both as a common compiler target language and as a macroassembler.

16.3.6 SISAL and Intermediate Format

The latest development to affect the project at the language level has been the design and implementation of the language SISAL (McGraw et al (26)) and its machine-independent target language IF (Skedzielewski and Glauert (27)). SISAL is another single-assignment language, based on Id and Val (Ackerman and Dennis (28)). It was designed by a consortium containing large computer users, a computer manufacturer, and academics with language design and implementation skills. The intermediate format IF is to be compiled onto several different types of multiprocessor

so that comparative evaluation of various approaches can be performed quickly and easily.

The SISAL/IF project is an exciting merger of wide-ranging interests. It is currently a major driving force for progress in the Manchester dataflow project. It is providing stimulus for hardware development in the form of an Overflow Unit for the Matching Unit, the 16 Megatoken Structure Store (see 16.2.6) and an Intelligent Token Queue for instruction scheduling. It is also providing impetus to the development of system software for program debugging, generating optimised code and performance tuning. IF is expected to replace TASS as the preferred assembly level language for the system.

16.4 TEST SYSTEMS

It will be appreciated that the complete dataflow system (hardware and software) has become highly complex. It is consequently difficult to test. In particular, the inherent parallelism of the system makes it difficult to isolate the cause of errant behaviour. We found ourselves developing system debugging tools by trial and error according to the nature of the faults under examination.

16.4.1 Hardware Test Programs

One of the best decisions in respect of engineering debugging turned out to be the use of a standard inter-module interface. As this was also the interface presented by the whole system to the outside world, it was possible to use the front-end system to test drive each module individually. Only when we were satisfied with the functionality of the component modules did we attempt to run programs on the complete system. Naturally we ran into problems due to the interaction of modules, but these were much easier to resolve given our basic confidence in the behaviour of the separate components.

However, once the system was interconnected, the mechanical task of splitting it back into modules turned out to be prohibitively lengthy. Consequently we have developed a set of remote test procedures which are capable of locating faulty modules and giving a certain degree of diagnostic information. Low-level test programs are constantly checking for consistent behaviour. The main pipeline data paths are periodically tested for continuity, and all load operations (for program code and function unit microcode) check for successful receipt of data by the appropriate module. Where modules themselves contain parallel components (eg in the Processing Unit) it is possible to isolate single boards for comprehensive testing. It is now rarely necessary to break the system into component parts to repair faults.

16.4.2 Software Fault-Tracing

Development of software during the project has introduced successively more 'layers' of program into the process of translation from application to machine. Whilst the translation programs themselves were being debugged, it was necessary to determine for each error where the fault occurred before tracing why. With up to five separate translation phases, each of which may rename or rearrange objects, this has been a great source of trouble. Some rationalisation of the translation process is currently taking place, and we have also introduced multi-level symbol table traces to facilitate error-finding.

16.4.3 Symbolic Debugging

Further improvement of the user interface to the system dictates that we implement symbolic debugging of programs from any input level. This is not as easy as it might be because of the parallel execution environment. Breakpoints have to act across multiple loci of control, and activation traces have to account for the asynchronous parallel nature of program execution. It would be preferable for programs to be proved before execution rather than debugged 'on the fly'. Consequently we have started theoretical studies which aim to formalise the semantics of dataflow programming. Ideas are still in their infancy in these areas, but we hope for important future breakthroughs from such work.

16.5 APPLICATIONS PROGRAMS

Although applications programs did play a part in producing the original architectural design, it was easy to forget their importance once engineering construction was underway. Consequently we found ourselves in the state that we had operational hardware with no firm idea of what we were going to use it for, or at least what we would use to evaluate its performance. This led to another trial and error phase during which benchmark programs were developed. It would be nice to report that we subsequently organised ourselves and arranged a systematic study of various applications areas in turn, but this is still a dream for the future. For now we have a random assortment of programs which have been brought to our attention by various sources and which we are trying to transfer to the dataflow hardware as quickly as possible.

16.5.1 Small Programs

The first programs to run on the machine were small examples of integer-manipulating codes. The best-known of these is the double-recursive factorial program which evaluates the factorial function by a parallel divide-and-conquer technique. For input values greater than 10 the factorial is greater than can be represented in a single-

length integer and so test runs used addition instead of multiplication, thus forming the sum rather than product of the first n integers. Another example was a nondeterministic version of the travelling salesman program which demonstrated the use of parallel execution to obtain the first available solution from a number of concurrent alternatives.

Other initial test programs used floating-point arithmetic and included trapezoidal integration, matrix multiplication and Gaussian elimination. A segment of a CAD code for VLSI geometry which calculates the location of a plumbline onto a polygon was also tested.

16.5.2 Physics Calculations

Self-contained example programs typical of large physics and signal processing calculations have been tried. The simplest programs were to calculate the Fast Fourier Transform and to solve a simple version of Laplace's equation using an iterative relaxation method. The largest code to be attempted so far has been the SIMPLE benchmark from Lawrence Livermore National Laboratory. Versions written in both Mad and SISAL have run on the hardware. Aerodynamic simulations based on solutions to the Navier-Stokes equations are being prepared and a Monte Carlo simulation is under development.

16.5.3 Computer-Aided Design

Some benchmarks for CAD programs have been written. A small combinational logic simulator and a Lee routing algorithm have been executed on the hardware. Larger programs concerned with simulation and consistency checking in VLSI systems are to be tested in the near future.

16.5.4 Standard Parallel Benchmarks

The random way in which we have collected benchmark programs highlights a general problem of comparing the performance of different multiprocessors. There is a strong case for the development of standard benchmark codes similar to those developed for conventional systems. However, the lack of a standard parallel programming language makes this difficult. It is hoped that the SISAL/IF project will go some way towards rectifying this problem, but there is still scope for much work in this area. An open question is whether standard Fortran benchmarks such as the 'Livermore Loops' can be parallelised for this purpose.

16.5.5 Program Characteristics

It was anticipated that certain characteristics of programs would influence the utilisation of the parallel hardware of the prototype dataflow system. Factors which have been recorded are the overall parallelism, the variation of parallelism over time, the proportion of each

type of instruction executed, the ratio of monadic to dyadic instructions and the source language used. Other factors, such as the amount of Matching Unit store required, can be expected to be influential, but these have not yet been systematically studied. Parallelism measurements have been made using the instruction-set emulator program which estimates the instantaneous parallelism by making simplifying assumptions about the way in which programs are executed on the hardware.

16.6 PERFORMANCE EVALUATION AND TUNING

The ultimate objective of an engineering research project like this is to come to some conclusions about the potential viability of the proposed system architecture by demonstrating its performance on a variety of benchmark programs. However, once the benchmark programs have been produced, there is more work to be done than merely report the results of program runs. It is usually possible to improve the system performance on each benchmark by 'tuning' the system hardware and software. If this must be done differently for each program then the architecture is not very general purpose. Of course, the designers hope that several such optimisations will be widely applicable and that they will improve performance continuously until it surpasses existing system capabilities.

We have only started to investigate the performance of the Manchester Dataflow system and we are not in a position to make definitive statements. The following sections outline the results we have accumulated so far.

16.6.1 Preliminary Performance Evaluation

Results of a preliminary evaluation of the prototype dataflow hardware are reported by Gurd and Watson (29). The results are based on hardware runs of a sample of the small applications programs mentioned in the previous section. Factors such as source language, overall parallelism and ratio of monadic to dyadic instructions were varied. Surprisingly, the results show that the sole influential parameter in determining the speedup characteristic for a program is its approximate overall parallelism as measured by the instruction-set emulator program. For details of this measurement, and further notes on performance evaluation, see the original paper by Gurd and Watson (29). The results are summarised by the curves (Fig.16.4).

16.6.2 Pipeline Tuning

As suggested by the preliminary performance evaluation, a pipeline buffer has been introduced into the ring between the Matching Unit and the Node Store. In this position, it reduces the inefficiency zone for highly parallel programs to about half the size discovered in the preliminary evaluation (29). This confirms that some of the reduced speedup is due to insufficient pipeline buffering, as was

speculated at the time, but more experimentation is still required. In particular, the impact of buffer size needs to be assessed, and the effect of positioning the buffer between the Node Store and the Processing Unit should be investigated.

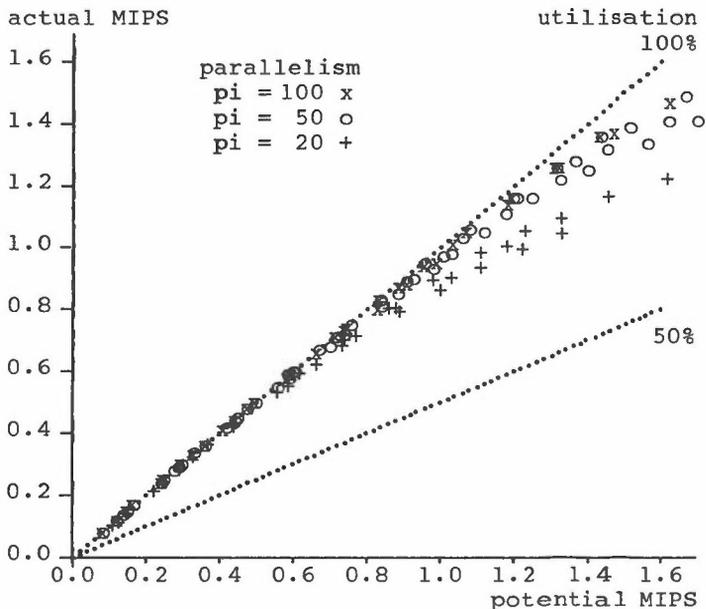


Fig.16.4 System speed versus number of function units

16.6.4 Future Work

The results reported above apply only to programs whose data sets are small enough to reside entirely within the Matching Unit store without hash table overflows occurring. This is an unrealistic requirement, and an evaluation of larger programs will be undertaken when hardware to handle overflows has been added to the system. Similar reasoning applies to the evaluation of the Structure Store extension to the architecture.

In addition, the above results are expressed in terms of dataflow MIPS, whose value compared to conventional MIPS is unclear. Experiments are in progress to establish this relationship. Initial results indicate that Manchester dataflow MIPS could be roughly comparable to VAX MIPS, given ideal compilation. In practice, current techniques produce several times as much code for the dataflow machine as they do for machines like the VAX.

16.7 CONCLUSIONS

Constructing a working prototype computer system is an exercise in faith and hard work. Events rarely work out as planned. Unforeseen problems occur, and unwanted kludges are introduced into the hitherto perfect design merely to get the thing going. The engineers always leave out some vital part which was difficult to build and whose purpose they couldn't understand. The troubleshooters need a huge breadth of knowledge in order to track faults through the myriad hardware and software layers. In the end the system takes so long to produce that it gets used for jobs that it was never intended to tackle.

Nonetheless, as usual with such obstacles, the satisfaction obtained from achieving the apparently impossible is immense. We hope we have conveyed the excitement and entertainment that construction of the Manchester prototype dataflow system has given us. If this paper has served to deter one or two trigger-happy computer architects from leaping into construction of their latest computer design then it has achieved one of its purposes. However, we would be disappointed if there were not at least one enthusiast who, having recognised the potential difficulties, and with due respect for the prospect of failure, nevertheless decided to take on the challenge of turning the ideas into reality.

ACKNOWLEDGEMENTS

None of the real engineering described in this paper would have been possible without the financial and organisational backing of the Distributed Computing Systems Programme of the Science and Engineering Research Council of Great Britain (under grants GR/A/74715, GR/B/40196 and GR/B/74788). Particular thanks for their support and helpful advice are due to Dave Aspinall, Iann Barron, Fred Chambers, David Duce, Keith Hanna, Bob Hopgood, Robin Milner, Roger Needham, Roger Newey, Charlie Portman, Mike Rogers and Rob Witty.

Research of this nature requires the efforts of a large and dedicated team. It is a pleasure to acknowledge the many contributions made to this project by the past and present members of the Manchester Dataflow Research Group: Val Aspin, Pedro Barahona, Karim Benkherouf, Wim Bohm, Dave Bowen, Alan Bradshaw, Ruth Brimley, Vicky Bush, Arthur Catto, Naranker Dulay, Mick Edwards, Greg Egan, John Foley, John Glauert, Ian Horrocks, Rob Jarratt, Peter Jinks, Katsura Kawakami, Geoff Lane, Jose Oliveira, Adrian Parker, Dave Pearson, Chris Richardson, Geoff Richmond, Carlos Ruggiero, John Sargeant, Bryan Saunders, Jose da Silva, Phil Treleaven, Sak Wathanasin, Pete Whitelock and John Zurawski.

REFERENCES

1. Lincoln, N., 1977, 'It's Really Not As Much Fun Building A Supercomputer As It Is Simply Inventing One', in: Kuck, D.J., et.al. (eds.), 'High Speed Computer and Algorithm Organisation', Academic Press, New York, USA.
2. Gurd, J.R., and Treleaven, P.C., 1976, 'A Highly Parallel Computer Architecture', Dept. of Comp. Sci., Univ. of Manchester, UK.
3. Treleaven, P.C., 1977, 'Exploitation of Parallelism in Computer Science', Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
4. Gurd, J.R., Watson, I., and Glauert, J.R.W., 1978 (1st edition) and 1980 (3rd edition), 'A Multilayered Dataflow Computer Architecture', Dept. of Comp. Sci., Univ. of Manchester, UK.
5. Egan, G.K., 1980, 'A Decentralised Computing System Based on Dataflow', Proc. IEEE Ind. Control and Instr. Conf., USA.
6. Arvind, Gostelow, K.P., and Plouffe, W., 1978, 'An Asynchronous Programming Language and Computing Machine', Tech. Rep. TR114a, Dept. of Info. and Comp. Sci., Univ. of California, Irvine, USA.
7. Dennis, J.B., 1974, Lect. Notes in CS, 19, p362.
8. Gurd, J.R., and Watson, I., 1980, Comp. Design, 9 6, p91.
9. Glauert, J.R.W., 1978, 'A Single-Assignment Language for Dataflow Computing', M.Sc. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
10. Gurd, J.R., and Watson, I., 1980, Comp. Design, 9 7, p97.
11. da Silva, J.G.D., and Watson, I., 1983, Proc. IEE, 130E 1, p19.
12. Bowen, D.L., 1981, 'Implementation of Data Structures in a Dataflow Computer', Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
13. Misunas, D.P., 1975, 'Structure Processing in a Dataflow Computer', Proc. Conf. on Parallel Computation, Sagamore, USA.
14. Arvind and Thomas, R.T., 1980, 'I-Structures : An Efficient Data Type for Functional Languages', Tech. Memo TM178, Lab. for Comp. Sci., MIT, USA.

15. Whitelock, P.J., 1978, 'A Conventional Language for Dataflow Computation', M.Sc. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
16. Veen, A.H., 1981, Lect. Notes in CS, 111, p127.
17. Kuck, D.J., et.al., 1981, 'Dependence Graphs and Compiler Optimisations', Proc. 8th ACM Symp. on Principles of Prog. Languages, USA.
18. Chamberlin, D.D., 1971, Proc. AFIPS, 39, p263.
19. Ashcroft, E.A., and Wadge, W.W., 1977, Comm. of the ACM, 20 7, p519.
20. Tesler, L.G., and Enea, H.J., 1968, Proc. AFIPS, 32, p403.
21. Bush, V.J., 1979, 'A Dataflow Implementation of Lucid', M.Sc. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
22. Sargeant, J., 1982, 'Implementation of Structured Lucid on a Dataflow Computer', M.Sc. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
23. Turner, D.A., 1976, 'SASL Language Manual', Dept. of Comp. Sci., Univ. of St. Andrews, UK.
24. Richmond, G., 1982, 'A Dataflow Implementation of SASL', M.Sc. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
25. Catto, A.J., 1981, 'Nondeterministic Programming in a Data Driven Environment', Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Manchester, UK.
26. McGraw, J., et.al., 1983, 'SISAL - Streams and Iteration in a Single-Assignment Language', Lawrence Livermore Nat. Lab., Cal., USA.
27. Skedzielewski, S., and Glauert, J.R.W., 1983, 'IF1 - An Intermediate Form for Applicative Languages', Lawrence Livermore Nat. Lab., Cal., USA.
28. Ackerman, W.B., and Dennis, J.B., 1979, 'VAL : A Value-Oriented Algorithmic Language - Preliminary Reference Manual', Tech. Rep. TR218, Lab. for Comp. Sci., MIT, USA.
29. Gurd, J.R., and Watson, I., 1983, 'Preliminary Evaluation of a Prototype Dataflow Computer', Proc. 9th IFIP World Comp. Congress, Tokyo, Japan.

Chapter 17

Shells of functional operating systems

P. Henderson, S. B. Jones

17.1 INTRODUCTION

We consider the shell of an operating system as the component responsible for structuring the interaction with the system user. As such we consider it a function from the sequence of items which we type to the sequence of items which appear on the screen. This view can be maintained even for quite elaborate shells which allow access to a database of files and allow for the composition of user-defined functional programs. Consequently we are able to build shells which are themselves proper functions and while simple enough to understand they are demonstrably powerful enough to support useful work. In this paper we develop a sequence of progressively more elaborate shells.

17.2 SIMPLE INTERACTION

Consider a program which accepts a sequence of numbers as input and displays a running total. Interaction with such a program might produce a screen with the following appearance:

$$\begin{array}{r} 3 \\ 3 \\ 4 \\ 7 \\ \hline 12 \\ 19 \\ -2 \\ \hline 17 \end{array}$$

The user's input is underlined. The screen's appearance is a function of the sequence of numbers the user has typed. This function has the following form:

```
totalise(total)(kb) ≡
  if kb = NIL then NIL else
    { let number = head(kb)
      kb' = tail(kb)
      cons(number,
            cons(total+number,
                  totalise(total+number)(kb')) ) }
```

We see that

```
totalise(0)((3 4 12 -2))
  = (3 3 4 7 12 19 -2 17)
```

A demand driven implementation of a functional programming language, determined to print to the screen every item of output as soon as it can be computed, will evaluate this function in interaction with the user in the appropriate way.

Imagine such an implementation which can evaluate any function of type

keyboard → screen

being asked to evaluate `totalise(0)`. Assume for the moment that the function perceives both keyboard and screen as a list of integers.

The implementation is determined to print a sequence of integers to the screen. Its determination cannot however overcome the test `kb=NIL?` in the definition until the user has completely entered his first number. Suppose the user types the number 3, terminated by a return. The implementation can now determine that the screen must have the form `(3 3.s)` where `s = totalise(3)`. Consequently it can print the first two items, the echoed input and the running total. Similarly, when the user types his second number 4, the next two items on the screen can be determined and printed.

A slightly different interaction would have occurred had we defined

```
totalise'(total)(kb) ≡
  cons(total,
        if kb=NIL then NIL else
          { let number = head(kb)
            kb' = tail(kb)
            cons(number, totalise'(total+number)(kb')) }
```

Now the initial running total 0 will appear before any action on the user's part, after which the interaction would be as before. This formulation shows how a prompt might be incorporated in an interactive functional program.

17.3 SEQUENCES OF INTERACTION

It is not enough to be able to run our `totalise(0)` function once. To run it repeatedly we introduce our first and simplest shell.

Here we will make use of two operations upon lists:

`untilend(x)` yields the prefix of `x` up to but not including the first occurrence of the word "end".

`afterend(x)` yields the suffix of `x` after the first occurrence of the word "end".

They are defined as follows.

```
untilend(x) ≡
  if x = NIL then NIL else
  if head(x) = "end" then NIL else
  cons(head(x), untilend(tail(x)))
```

```
afterend(x) ≡
  if x = NIL then NIL else
  if head(x) = "end" then tail(x) else
  afterend(tail(x))
```

Assume `f` is a function of type `keyboard → screen`. Then `repeat(f)` is also a function of type `keyboard → screen` defined as follows.

```
repeat(f)(kb) ≡
  if kb=NIL then NIL else
  append(f(untilend(kb)), repeat(f)(afterend(kb)))
```

It is easy to see that `repeat(f)(kb)` applies `f` to each subsequence of `kb` delimited by "end"s. For example `repeat(totalise(0))((3 4 end 7 -2 19 end 6 -10))`

```
=(3 3 4 7
  7 7 -2 5 19 24
  6 6 -10 -4)
```

where the three separate interactions with `totalise(0)` each commence with the running total at 0.

Not all interactive programs are conveniently terminated by "end" however. Since it proves inconvenient to impose upon the user the discipline of segmenting the input using "end"s we introduce an alternative device for describing the repeated evaluation of a user-defined function. Instead of considering an interactive program to be a function of type `keyboard → screen` we require it to be of type

`keyboard → screen x keyboard`

This extra result is expected to be the unconsumed suffix of the keyboard handed to it as an argument. A very simple function of this type is

```
double(kb) ≡
  if kb = NIL then (ERROR), NIL
  else {list(number, 2*number), kb'
        where number = head(kb)
              kb'     = tail(kb)}
```

This function is intended to double the one and only number presented to it as an argument.

Consider how we might evaluate double.

Define

```
interact      = keyboard → screen
reminteract   = keyboard → screen x keyboard
```

to denote the two different types of interactive function we have defined. The shell repeat is of type $\text{interact} \rightarrow \text{interact}$. Let us define remrepeat of type $\text{reminteract} \rightarrow \text{interact}$ as follows

```
remrepeat(f)(kb) ≡
  let scr', kb' = f(kb)
  append(scr', if kb' = NIL then NIL
           else remrepeat(f)(kb'))
```

The only unusual property of this shell is the way that termination is handled.

If we evaluate $\text{remrepeat}(\text{double})((2\ 7\ -1))$ then the result will be $(2\ 4\ 7\ 14\ -1\ -2)$. To see this, let us construct a reduction sequence which will demonstrate how termination is dealt with properly.

```
remrepeat(double((2 7 -1))) =
append((2 4), remrepeat(f)((7 -1))) =
append((2 4), append((7 14), remrepeat(f)((-1)))) =
append((2 4), append((7 14), append((-1 -2), NIL))) =
(2 4 7 14 -1 -2)
```

Now it is possible of course to use remrepeat to execute totalise(0). First we define a function which makes an interact into a reminteract as follows. Assume f is of type interact, then endwrap(f) is of type reminteract.

```
endwrap(f)(kb) ≡ f(untilend(kb)), afterend(kb)
```

Now, it is the case that

```
remrepeat(endwrap(totalise(0)))
```

has the same interactive behaviour as

```
repeat(totalise(0))
```

17.4 INTERACTION WITH DATABASES

Consider a simple (flat) file system which consists of an array of files indexed by file names. We shall denote such a file system by a finite function (association list, array) whose domain is file names and whose range is file contents. We will use the following notation. If db is a file system and fn is a file name in the domain of db then we denote the corresponding file contents by $db[fn]$. If fn_1, fn_2, \dots are distinct file names, and fc_1, fc_2, \dots are the corresponding file contents then we denote the file system which contains exactly these associations by $\{fn_1 \rightarrow fc_1, fn_2 \rightarrow fc_2, \dots\}$.

If db and db' are file systems then $db@db'$ is a file system which contains all of the associations of db and of db' except where a file name is in the domain of both db and db' . Then only the association from db' is recorded. Hence

$$(db@db')[fn] = \begin{array}{l} \text{if } fn \text{ in } \text{dom}(db') \text{ then } db'[fn] \\ \text{else } db[fn] \end{array}$$

We want to build a shell which will allow interaction with programs which can also access the file system. Let us define a new form of interaction which is like `reminteract` except that it includes access to the file system

```
dbinteract = keyboard x database →
              screen x keyboard x database
```

A simple example of such an interactive function is the following:

```
copy(fn1,fn2)(kb,db) ≡
  NIL,kb, if fn1 in dom(db) then db@{fn2 → db[fn1]}
  else db
```

This particular version of `copy` requires no input from the keyboard and produces no output on the screen. If there is a file called fn_1 then its contents are copied to fn_2 . This may cause fn_2 to be overwritten.

Another example lists a file contents to the screen

```
from(fn)(kb,db) ≡
  (if fn in dom(db) then list(db[fn]) else NIL),kb,db
```

On this occasion no input is required and the database is left unchanged. Similarly we can define a function which directs a prefix of the keyboard into a file.

```
to(fn)(kb,db) ≡
  untilend(kb),afterend(kb),db@{fn → until end(kb)}
```

We have reflected the keyboard on the screen, placed everything until "end" in the file fn and returned the unconsumed portion of the keyboard.

Consider how we might compose such functions in order to benefit from the accumulated evaluation of two or more of them. Let us define `compose(f,g)` to be of type `dbinteract` whenever `f` and `g` are both of that type.

```
compose(f,g)(kb,db) ≡
  append(scr',scr"),kb",db"
  where scr',kb',db' = f(kb,db)
        scr",kb",db" = g(kb',db')
```

Now we can see that `compose(from(fn1),from(fn2))` lists both `fn1` and `fn2` in that order, that `compose(copy(fn1,fn2),from(fn2))` copies one file to another and then lists the new file and that `compose(to(fn1),from(fn1))` creates (or overwrites) a file and lists its contents. Longer compositions are possible. For example,

```
compose(copy(fn1,fn2),compose(to(fn1),
  compose(from(fn1),from(fn2))))
```

first saves `fn1`, then overwrites it from the keyboard and finally lists both files to check their contents.

For completeness, let us build a repeating shell similar to those we have already built. This time `dbrepeat` expects an argument `f` of type `dbinteract`. Note that `dbrepeat(f,db)` is of type `interact` (i.e. keyboard → screen) where `db` is some initial database.

```
dbrepeat(f,db)(kb) ≡
  let scr',kb',db' = f(kb,db)
  append(scr', if kb' = NIL then NIL
        else dbrepeat(f,db')(kb'))
```

Termination is handled in the same way as for `remrepeat`. We see that evaluation of

```
dbrepeat(compose(to(fn),from(fn)),NIL)
```

will, if presented with a suitable argument, repeatedly refill the file `fn` and list it. A fairly pointless activity. But we will use `dbrepeat` later to build a more useful shell.

Given a function `g` of type `reminteract` we can elevate it to a function of type `dbinteract` trivially using

```
dbwrap(g)(kb,db) ≡
  let scr',kb' = g(kb)
  scr',kb',db
```

So for example `dbrepeat(dbwrap(double),db)` has the same meaning as `remrepeat(double)` regardless of the value of `db`.

17.5 INTERACTING WITH A DATABASE OF PROGRAMS

Finally we come to the purpose of this whole exercise. We now have the basic machinery required to build a realistic shell which allows the user to create, manipulate and evaluate programs of his own design. We shall build only the simplest such shell. We require to store programs of type `dbinteract` in the database, to load them and to evaluate them.

We assume that the user will type commands of the form `(f args)` where `f` is the name of a file in the database which holds a "loadable" program. The file contents, which are text created by an editor or a compiler, when loaded become a function of type `args → dbinteract`. Hence we assume the existence of a function

```
load:text → (args → dbinteract)
```

Consider now the form in which a program must be prepared to be acceptable to load. For example `copy` can be redefined as

```
COPY(args)(kb,db) ≡
    copy(arg1(args),arg2(args))(kb,db)
```

where `arg1` and `arg2` select the first and second arguments from the command-line handed to `COPY` by the shell. We assume the file called `COPY.COM` contains the text of this function. The shell will then construct `load(COPY.COM)` which we assume to be the function `COPY`. If the command line `(COPY.COM(FRED JOE))` has been supplied, the shell will copy file `FRED` to file `JOE`.

Consider first a function which executes a single command. For simplicity we pay no attention to error cases.

```
execute(command)(kb,db) ≡
    let program = first(command)
        args     = second(command)
    load(db[program])(args)(kb,db)
```

Here we assume the command has the form `(program args)`. We seek the text of the program from the database and "load" it. We apply the "loaded" program to its arguments to yield a function of type `dbinteract`. This function can be applied to `kb` and `db` to produce the required triple of results.

This demonstrates that `execute(command)` is of type `dbinteract`.

Now, if we define

```
shellstep(kb,db) ≡
    if kb = NIL then NIL,NIL,db
    else execute(head(kb))(tail(kb),db)
```

we see that shellstep is also of type dbinteract. It executes exactly one command.

So finally we can define our shell. The function shell is of type interact.

```
shell ≡ dbrepeat(shellstep,NIL)
```

If we have recorded the functions

```
TO(args)   ≡ to(arg1(args))
FROM(args) ≡ from(arg1(args))
COPY(args) ≡ copy(arg1(args),arg2(args))
```

in the files TO.COM, FROM.COM and COPY.COM respectively the shell should be able to participate in the following interaction.

```
(TO.COM (FRED))
VARIOUS LINES
OF
DATA
end
(FROM.COM (FRED))
VARIOUS LINES
OF
DATA
(COPY.COM (FRED JOE))
(FROM.COM (JOE))
VARIOUS LINES
OF
DATA
```

17.6 CONCLUSIONS

We have attempted to illustrate how an interactive shell for an operating system can be built as a composition of purely functional programs. We have evaluated these ideas by implementing a series of shells similar to the simple examples described here. Problems which remain to be adequately solved are many. We have not discussed the problems which arise with non-determinacy which are manifest in many of the operations of a system shell. We have not dealt with the important issue of error handling. But we do believe we have made some impression upon the problem of dealing with interaction in a purely functional style. The advantages of doing so, which are the potential for parallel evaluation and the possibility of demonstrating correctness by mathematical argument, are yet to be realised.

REFERENCES

1. Henderson, P, 1982, Functional Programming - Application and Implementation, Prentice Hall International, ISBN 13-879999-7.
2. Henderson, P, 1982, 'Purely Functional Operating Systems', in Functional Programming and its Applications, eds. Darlington, Henderson, Turner, Cambridge University Press.
3. Jones, S B, 1983, 'Abstract Machine support for Purely Functional Operating Systems'. Technical Monograph PRG-34, Programming Research Group, Oxford University.
4. Jones, S B, (forthcoming), 'A range of operating systems written in a purely functional style'. (To appear as a technical report).
5. Clark, K L, and Gregory, S, 1981, 'A Relational Language for Parallel Programming', Imperial College.
6. Clark, K L, and Gregory, S, 1983, 'PARLOG: A Parallel Logic Programming Language', Research Report DOC 83/5, Imperial College.
7. Shultis, J, 1983, 'A Functional Shell', ACM SigPlan Notices, Vol.18, No. 6, 202-211.

Index

- Abstract data types 64, 224
- Abstraction
 - Data 62
 - Process 154-
- Active memory array 187-
- Ada 40, 47-, 68, 88, 91, 121, 263
 - task entries 43
 - task procedure 43
- Adaptive control algorithms 103
- Address staggering 212
- Adequacy 116
- Afterend 292
- Algorithm
 - Analysis 169-
 - Design 169-
- ALICE 8, 221, 238
- Alto 203
- Alvey Programme 4, 9
- Amsterdam Compiler kit 104
- Analysis of concurrent systems 108-
- Animation 199-
- Annotations 256
- Annual Report 5
- Applicative languages
 - see Languages
- Array manipulation 194
- Array Processor 207-
- Arrays of microprocessors 53
- Assembly languages
 - see Languages
- Associative store 277
- Atomicity 104
- Axiomatic semantics
 - see Semantics
- Backtracking 224
- Barrel shifter 215
- Baseband networks
 - see Local area networks
- Basic 58
- Basic block protocol
 - see Protocols
- Basic COSY system
 - see COSY
- Basix 58
- BCPL 40, 53, 55
- Behaviours 108, 264
- Benchmarking 284
- Beta reduction
 - see Reduction
- Birth and death processes
 - 1-Dimensional 142
 - 2-Dimensional 143
- Bit-stuffing 35
- Block triangular method 172
- Breadth-first 253, 266
- Broadband Data Networks
 - see Local area networks
- Buffered message passing
 - see Message passing
- C 41, 55
- CAD tools 277, 284
- Cambridge Ring
 - see Local area networks
- Capabilities 192-
- Carrier sense multiple access
 - see CSMA
- Catalog approach 227
- CATV 12, 14
- Causal nets
 - see Nets
- CCS 121, 263
- Centrenet
 - see Local area networks
- Channel 43
- Church-Rosser theorem 228, 251

300 Index

- Clarity 3
- Class 68
- Closely-coupled systems 7
- Closures 253
- Coaxial cable network 12
- Combinators
 - see Reduction
- Communication 264
 - Synchronised 52
 - Unbuffered 52
- Communication mechanism 41
- Communication primitives 90
- Community Antenna Television
 - see CATV
- Complexity 176-
- Composition 134
- Computer aided teaching 73
- Computer Science committee 1
- Concurrency 8, 39, 107-
- Concurrent Pascal
 - see Pascal
- Concurrent reachability
 - see Reachability
- Concurrent systems 107
- Condition/event nets
 - see Nets
- Conferences 5
- Configuration
 - Database 98
 - Language - see Languages
 - Manager 97-
 - Management 96-
- Conic 40, 50-, 58, 86-
 - Fault tolerant Conic 104
- Cons 291
- Content-addressable memory 210
- Coordinate transformations 217
- Coordinated programme 3
- Coordination 3-
- Coral 41, 57
- Coroutines 57, 165
- Correctness 226-
 - Partial 227
 - Total 227
- Cost 3
- COSY 107-
 - Applications 119
 - Basic COSY System 118
 - Dossiers 118
 - High level notation 118
 - Macro notation 118
 - Path programs 111
 - Semantics 118, 120
 - System notation 118
 - VLSI implementation 119
- CRAY 169
- CSMA 20
- CSP 53, 88, 121, 263, 281
- Cube-connected cycles 256
- DAP 169, 188-, 207
- Data abstraction
 - see Abstraction
- Data bases 63, 294-
 - Distributed 141, 148-
- Data structure store
 - see Structure store
- Data transmission system 12
- Dataflow 10
- Datagram 37
- DDA algorithm 204, 217
- Deadlock-free 107, 116
- Debugging 101, 225, 282-
- Declarative languages
 - see Languages
- Declarative systems 9
- Demand drive 280, 291
- Demand forking 256
- Denotational semantics
 - see Semantics
- Depth-first 253
- Design methodologies 8, 108-, 154-
- Development of concurrent systems 108
- Directed programme 3
- Directors 262-
- DisArray 199-
- DisArray2 215-
- Display list 199, 201-
- Distributed Computing Systems Programme (DCS) 2-
- Distributed data bases
 - see Data bases
- Distributed data store
 - see Structure store
- Distributed filestores
 - see Filing system
- Distributed operating systems
 - see Operating systems
- Distribution 3
- Distribution of clocks 276
- Divide and conquer algorithms 250, 255-, 283
- DP 281
- DTL 58, 154-
- Duobinary AM-PSK 21
- DyNe 256, 265-

- Edison 58, 121
- Eigenvalue problem 170
 - see also Tridiagonal eigenvalue problem
- Equipment pool 5
- Ethernet
 - see Local area networks
- Evaluation order 228-
- Explicit numerical methods 181
- Extended semaphore primitive 122

- Fair merge
 - see Merge
- Fairness 230
- Fast Fourier transform 284
- Fault tolerant Conic
 - see Conic
- Fibernet
 - see Local area networks
- Fibre-optics 27-, 34-
- Filing system 56, 62, 101, 294-
 - Distributed 9
 - Specification 126-
- Finite state automata 109
- Firing sequences 112-
 - see also Vector firing sequences
- Fixed Frequency Modem
 - see Modem
- Fixed-point equations 141, 147
- Flexibility
 - Functional 86
 - Implementation 86
 - Time domain 87
 - Topological 87
- Flexible Manufacturing Systems 56
- Folding 226
- Fork 40
- Fork-join 40, 47
- FORTRAN 279
- FP 227
- Frequency agile modem
 - see Modem
- Frequency modulation 35
- Frequency synthesizer 17, 19
- Frequency-division-multiplexing 14
- Functional flexibility
 - see Flexibility
- Functional languages
 - see Languages
- Functional operating systems
 - see Operating systems
- Functional programming
 - see Programming
- Functions 290

- G-machine 237
- Garbage collection 238-, 278
- Gaussian elimination 284
- General net theory
 - see Nets
- Generalised periods 113
- Generative set approach 227
- Global clock 43
- Global synchroniser 45
- Graph reduction
 - see Reduction
- Graph reduction machines
 - see Reduction
- Graph rewriting 261
- Group modules 94
- Guarded commands 43, 53, 55, 57, 91
- Guardians (Argus) 96
- Guards
 - see Guarded commands

- Hardware description languages 161-
- Hardware failure 46
- HDLC 34
- Heat-conduction problem 181
- Hiatons 231
- Hierarchical decomposition 155
- High Bit Rate Modem
 - see Modem
- High integrity design 190-
- High level COSY notation
 - see COSY
- High level transformations 227
- Higher order functions 225, 236
- History 108
- Hope 221-, 252

- Id 280-
- Idempotent messages 50
- IEEE-488 36
- IF 281-
- Imperative languages
 - see Languages
- Implementation flexibility
 - see Flexibility
- Implicit numerical methods 181
- Independence relations 114
- Infinite data structures 229
- Infrastructure 4-
- Innermost spine reduction

302 Index

- see Reduction
- Instantaneous transition rates 140
- Interaction 292-
- Interactive functional programming 291-
- Interconnection topology 260-
- ISWIM 252

- Jackson networks 144
- Jackson program design method 159
- Join 40

- Kleinrock's conservation law 144
- KRC 8, 221-

- Labelled Petri nets
 - see Nets
- Labelled-token dataflow 270-
- LAMBDA 232
- Lambda calculus 232, 252, 256, 261
- Languages
 - Applicative 8, 220-, 279
 - Assembly 281
 - Configuration 54, 88-, 92-
 - Declarative 220-
 - Functional 154, 220-, 250, 260, 279, 290-
 - Imperative 279
 - Logic 8, 252, 279
 - Non-deterministic 279
 - Non-procedural 279
 - Single assignment 279
 - Specification 107-, 127-, 228
- Laplace's equation 284
- Lapse 275, 280
- Lazy evaluation 220, 225, 229
- LCF 221
- Lee routing algorithm 284
- Line drawing 204, 217
- Lisp 58
- Lispkit 221, 252
- Little result 141
- Livelock 45
- LNET 256, 264-
- Loaders 101
- Local area networks 12-, 25-, 94
 - Baseband 13, 14
 - Broadband 14-
 - Cambridge Ring 6, 10, 26, 58, 94, 103, 149
 - Centrenet 25-
 - Ethernet 13, 26, 94, 150
 - Fibernet 27
 - Modelling 141, 149-
 - Sussex network 12-
 - Token ring 13, 151-
- Local link 35-
- Logic languages
 - see Languages
- Logic networks 161
- Logic programming
 - see Programming
- Loosely-coupled systems 7
- Lucid 280

- M/M/N system 142
- Macro notation (COSY)
 - see COSY
- Mad 271, 280
- Mailboxes 43, 55
- Mailshot 5
- Management of DCS 3
- Manchester dataflow machine 8, 267, 270-
- Manchester encoding 21
- Markov processes 139-
 - N-dimensional 144
- Markov property 139-
- Martlet 40, 45, 47-
- Matching functions 278, 281
- Matching unit 275, 277-
- Mathematical modelling 139-
- Matrix factorisation 185
- Matrix multiplication 284
- Maximal concurrency 115
- Maximal concurrent evolution 115
- Maximally concurrent reachability
 - see Reachability
- Mean Value analysis 148
- Mediums 50, 58
- Meetings programme 5
- Merge
 - Fair 230
 - Non-deterministic 229
- Merge algorithm
 - Parallel 176
- Message passing 88-
 - Buffered 41-
 - Synchronised 41-
 - Unbuffered 41-
- Meta language 227
- MINAS operating system 43
- Miranda 8, 221
- ML 221-, 252
 - Standard 221-
- Modem

- Fixed frequency 17-
- Frequency agile 17-
- High bit rate 20-
- Modula 91
- Modula-2 40
- Modularity 88, 134
- Module 50, 105, 224
- Monitors 41, 46, 56, 88, 122
- Monte Carlo simulation 284
- MU5 26
- MU6 26
- Multi-ring dataflow system 271, 278-
- Multi-window screen interface 56
- Mutual exclusion 71

- N-dimensional Markov processes
 - see Markov processes
- Navier-Stokes equation 284
- Neighbour sort algorithm
 - see Sorting
- Nets
 - Causal 109
 - Condition/event 109
 - General net theory 109
 - Labelled Petri 120
 - Occurrence 109
 - Petri theory 109
- Network Architectures 26-
- Network for University Campus 12, 26
- Network Intelligence Module 29
- Network models 144
- Network protocols
 - see Protocols
- Network reconfiguration 52
- Node store 275
- Non von-Neumann architectures 7
- Non-determinism 43, 107, 229, 297
- Non-deterministic languages
 - see Languages
- Non-deterministic merge
 - see Merge
- Non-procedural languages
 - see Languages
- Normal order reduction
 - see Reduction
- Notify ports 51, 88
- NPL 223
- NRZI-S coding 35

- OBJ 63
- occam 40, 53-, 263
- Occurrence nets
 - see Nets
- Omninet 103
- Open systems 126
- Operating systems
 - Distributed 9, 99-, 126-
 - Functional 9, 290
- Optical fibre transmission
 - see Fibre-optics
- Optimal reduction orders
 - see Reduction
- Oracles 231
- OS6 134
- OSI model 37
- Overflow unit 278

- P-machine 63
- Painter's algorithm 201
- par 235
- Parallel
 - Arithmetic 191
 - Bisection 171, 174
 - Conditionals 231-
 - Graph reduction - see Reduction
 - Hashing scheme 277
 - Merge algorithm - see Merge algorithm
 - Multisection 174
- Partial correctness
 - see Correctness
- Partial functions 127
- Partitioning 169
- Pascal 40, 46, 63, 64, 279
 - Concurrent 68, 156
 - Pascal Plus 40, 46, 56-, 68
 - Pascal-m 9, 40, 55-
 - Path Pascal 41, 46, 56-
 - UCSD 63
- Path expressions 57
- Pattern matching 222-
- PCF 232
- Performance 3, 8, 10, 102, 169, 179-, 196-, 214-, 285
 - Modelling 139-
- Perq 63, 203
- Persistent data structures 82
- Petri net theory
 - see Nets
- Phase encoding 35-
- Plant monitoring 63
- PO 279
- Pointer-number machine 189-
- Poisson distribution 144-
- Polygon filling 217

- Polymorphic typing 220, 222-
- Ponder 222
- Pop-up menus 199
- Port-card 30-
- Ports 43, 51, 88-
- Powerdomains 231
- Process 40, 54, 253
- Process abstraction
 - see Abstraction
- Process control 50-
- Process tree 252
- Processing unit 275
- Program development environment
 - 72-
- Program proving 226
- Program structuring 40
- Program transformation 225-
- Programming
 - Functional 224-, 290-
 - Interactive functional 291-
 - Logic 224-
- Prolog 224
- Protocols 45
 - Basic block 150-
 - Specification 119
 - Verification 119
- Pulse project 47-, 50, 58

- QMC Text terminal 203
- Quadrant addressing 213
- Quadrant interlocking factorisation 185
- Quadrature 170-
- Queueing network 151
- Queueing theory 139-
- Quicksort 234

- r-n-cube 254, 256
- RasterOp 200-
- Reachability 114
 - Concurrent 115
 - Maximally concurrent 115
 - Sequential 115
- Recursion 53
- Recursive data structures 64
- Recursive doubling 171
- Redex 228
- Reduction 251-
 - Beta 261
 - Combinator 8, 233, 238, 255, 262-
 - Graph 10, 232-, 265
 - Innermost spine 233
 - Machines 237-
 - Normal order 228-
 - Optimal order 233
 - Parallel graph 263-
 - Safe order 267
- Referential transparency 226
- Refresh controller 211
- Regular expressions 111
- Regular grammars 109
- Reliability 3
- Remote links 33
- Remote procedure call 41-, 46, 47, 58
- Rendezvous mechanism 49
- Request-reply ports 52, 88
- Rewrite rules 251
- Rewrite systems 260-
- RISC 190, 195
- Routing matrix 144

- Safe reduction orders
 - see Reduction
- SASL 8, 221-, 252, 261, 280
- Schema (Z) 129
- SECD machine 238
- Semantics
 - Axiomatic 266
 - COSY - see COSY
 - Denotational 266
 - Operational 121
- Sequences 290-
- Sequential bisection 174
- Sequential reachability
 - see Reachability
- Sequential translations 159-
- SERC 1
- Servers 57
- Set notation 223
- Shared-memory 41, 46, 48, 51, 56, 169
- Shells 290-
- SIMPLE benchmark 284
- Single-assignment languages
 - see Languages
- Single-user system 13, 199
- SISAL 281-
- Sorting 157, 176
 - Neighbour algorithm 176
- Space complexity 234
- Specification 8, 9, 105, 108-, 126-, 154, 225, 228
 - Languages - see Languages
 - see also Protocols
- SR 91
- Stability properties of algo-

- rithms 174
- Standard ML
 - see ML
- Star-type network 27
- Starpoint 29-
- Starvation 108
- Steady-state balance equations 141
- Steady-state distribution 140
- Stepwise refinement 79, 154-
- Stochastic processes 139
- Store-and-forward gateways 94
- Strictness 235
- Structural operational semantics
 - see Semantics
- Structure editor 72-
- Structure store 63-, 274, 278, 281
- Structured data store
 - see Structure store
- Structured programming 154-
- Sturm sequence 171
- Supercombinators 237
- Supercomputer 270
- Superport 32
- Surface shifting 215-
- Sussex network
 - see Local area networks
- Switch 275-
 - sync 235
- Synchronisation 107-
- Synchronised message passing
 - see Message passing
- System notation (COSY)
 - see COSY
- Tag coding 192
- Tagged memory 190-
- Tagged-token dataflow 272
- Tasks 47, 50, 88-
- TASS 281
- Technology transfer 10
- Template assembler 271, 281
- Templates 264
- Terminal Switching Exchanges 13
- Termination 293-
- Textual substitution 238
- Theory 9
- Tightly-coupled systems 9
- Time domain flexibility
 - see Flexibility
- Time-out 45, 48, 152
- Timestamps 231
- Timing problems 44-
- Token matching unit 274
- Token queue 275
- Token ring
 - see local area networks
- Tokens 272-
- Topological flexibility
 - see Flexibility
- Total correctness
 - see Correctness
- Trace 108
- Train journeys 119
- Transformation systems 227-
- Transformational development of specifications 117
- Transitive closure 217
- Trapezoidal integration 284
- Travelling salesman 284
- Tridiagonal eigenvalue problem 171-
 - see also Eigenvalue problem
- Tripes 40
- Type editor 72-
- Type-checking 49, 57
- UCSD Pascal
 - see Pascal
- Unfolding 226
- Unification 224-
- Unix 6, 9, 58
- Unix United 9
- Untilend 292
- User interfaces 72, 199
- Val 281
- Value editor 79-
- Vector
 - Events 113-
 - Firing sequences 113-, 116-
 - Maximal firing sequences 115
 - see also Firing sequences
- Vectorization 169
- Verification 105, 108-
- Video data 37
- Virtual memory 62-
- Voice transmission 37
- Wafer scale integration 260
- Wide area network 25
- Windows 199-
- Z specification notation 127-
- ZAPP 250-
- Zermelo-Frankel set theory 222-