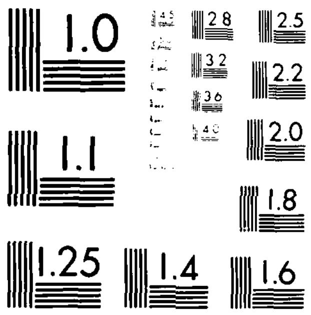
AD-A145 134   MIT LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 20   1/3
- JULY 1982 - JUN 1983(U) MASSACHUSETTS INST OF TECH
CAMBRIDGE LAB FOR COMPUTER SCIENCE M L DERTOUZOS
UNCLASSIFIED   JUL 84 N00014-83-K-0125   F/G 9/2   NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

1

PROGRESS REPORT 20

July 1982 - June 1983

Prepared for the

Defense Advanced Research Projects Agency

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5896

84

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS Progress Report 20	2. GOVT ACCESSION NO. A145134	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MIT Laboratory for Computer Science Progress Report 20 - July 1982-June 1983		5. TYPE OF REPORT & PERIOD COVERED Progress Report 1982-83
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael L. Dertouzos, Principle Investigator, MIT Lab for Computer Science		8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0125 DARPA/DOD
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA, IPTO 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE July 1984
		13. NUMBER OF PAGES 225
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR/Dept. of the Navy Information Systems Division Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution is Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for Public Release; Distribution Unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

		Theory
Computation Structures	Dataflow	Multiprocessing
Computer Architectures	Distributed Systems	Personal Computers
Computer Languages	Educational Computing	Programming
Computer Networks	Hardware Systems	Real-Time
Computer Systems	Local Networks	VLSI

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The work reported herein was carried out within the MIT Laboratory for Computer Science during 1982-1983. This report summarizes the activities performed under DARPA funding.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PROGRESS REPORT 20

July 1982 - June 1983

Prepared for the

Defense Advanced Research Projects Agency

Effective date of contract:	**1 January 1983**
Contract expiration date:	**31 December 1985**
Principal Investigator and Director:	**Michael L. Dertouzos** (617) 253-2145

This research is supported by the Defense Advanced Research Projects Agency under Contract No. N00014-83-K-0125. Views and conclusions contained in this report are those of the authors and should not be interpreted as representing the official opinions or policy, either expressed or implied of DARPA, the U.S. Government or any other person or agency connected with them. This document is approved for public *release and sale; distribution is unlimited.*

TABLE OF CONTENTS

[illegible]

ADMINISTRATION

Academic Staff

M. Dertouzos	Director
M. Rivest	Associate Director
A. Vezza	Associate Director

Administrative Staff

J. Hynes	Administrative Officer
P. Anderegg	Assistant Administrative Officer
M. Jones	Fiscal Officer

Support Staff

G. Brown	C. Morin
J. Coleman	E. Profirio
L. Cavallaro	M. Sensale
R. Cinq-Mars	J. Spillane
R. Donahue	C. Stevens
T. LoDuca	P. Vancini

Work reported herein was carried out within the Laboratory for Computer Science, an MIT interdepartmental laboratory. During 1982-83, the principle financal support of the Laboratory has come from the Defense Advanced Research Projects Agency (DARPA), under the Office of Naval Research Contract N00014-83-K-0125. DARPA has been instrumental in the last 20 years and is gratefully acknowledged here.

Final assembly and production of this report was done by P.A. Vancini.

INTRODUCTION

The MIT Laboratory for Computer Science (LCS) is an interdepartmental laboratory whose principal goal is research in computer science and engineering.

Founded in 1963 as Project MAC (for Multiple Access Computer and Machine Aided Cognition), the Laboratory developed the Compatible Time-Sharing System (CTSS), one of the first time-shared systems in the world, and Multics -- an improved time-shared system that introduced several new concepts. These two major developments stimulated research activities in the application of on-line computing to such diverse disciplines as engineering, architecture, mathematics, biology, medicine, library science, and management. Since that time, the Laboratory's objectives expanded, leading to research across a broad front of activities.

The first such area entitled Knowledge Based Systems, involves making programs more intelligent by capturing, representing, and using knowledge which is specific to the problem domain. Examples are the use of expert medical knowledge for assistance in diagnosis carried out by the Clinical Decision Making Group; the use of solid-state circuit design knowledge for an expert VLSI (very large scale integration) design system by the Real Time Systems Group; and the use of specific knowledge about budgets for an expert planning system by the Programming Technology Group.

Research in the second and largest area entitled Machines, Languages, and Systems, strives to discover and understand computing systems at both the hardware and software levels that open new application areas and/or effect sizable improvements in their ease of utilization and cost effectiveness. For example, the Programming Methodology Group and the Real Time Systems Group are developing languages and operating systems for use in large (thousands of computers) geographically distributed systems. The extended networks for such distributed environments are studied by the Computer Systems and Communications Group, while distributed file servers and cryptographic protection techniques are pursued by the Computer Systems Structure Group. New research in this overall area also includes the architecture of very large multiprocessor machines (dedicated to a single task, e.g., speech understanding or machine vision) by the Computation Structures, Functional Languages and Architectures, and Real Time Systems Research Groups.

The Laboratory's third principal area of research, entitled Theory, involves exploration and development of theoretical foundations in computer science. For example, the Theory of Computation Group strives to understand ultimate limits in space and time associated with various classes of algorithms; the semantics of programming languages from both analytical and synthetic viewpoints; the logic of programs; and the links between mathematics and the privacy/authentication of computer-to-computer messages. Another example of work in this area involves the study of distributed systems, by the Theory of Distributed Systems Research Group.

The fourth area of research entitled Computers and People, entails societal as well as technical aspects of the interrelationships between people and machines. Examples include the use of computers in the educational process by the Educational Computing Group; the use of interconnected computers for planning; as well as the societal impact of computers carried out by the Societal Implications Research Group.

During 1982-1983, the Laboratory embarked on the ambitious project of constructing an emulation facility consisting of 64 interconnected large computers, whose purpose is to analyze the behavior of larger (up to several thousand machines) multiprocessor systems. This facility will enable experimenters at MIT and elsewhere in the United States to try out their ideas before committing their proposed architectures into silicon circuits. Another important development during this period has been the development of CONCERT by Professor Robert Halstead of the Real Time Systems Group. It consists of a prototype four-processor multiprocessor system that will eventually interconnect 32 processors. Together with a parallel version of LISP, this system will be used primarily to explore the use of multiprocessor systems in graphics and other applications.

Another growth activity during 1982-83 has been the newly established Educational Computing Group which is headed by Professor Harold Abelson and includes Professor Seymour Papert, and Drs. Andrea diSessa and Sylvia Weir. This group, which in the last 12 years developed the widely used language LOGO, is currently focusing its efforts on the use of computer technology, cognitive science, and educational innovation, primarily at the secondary school level.

During this reporting period we have also made substantial progress in distributed systems research. This major Laboratory focus continues to occupy the attention of about half of our people. Our recent results have put us in a position to construct a class of geographically distributed and interconnected systems which strive to balance local autonomy with application cohesiveness. The hardware resources that we designed have been transferred to industry (Texas Instruments) and we expect to take delivery of 30 commercial-level advanced personal computers before the end of 1984. These and other related machines (single-user Vaxes and Lisp Machines) are being interconnected into prototype distributed systems within the Laboratory. It is through these prototypes that we are implementing the collection of research results that we have acquired up to now. In particular, we are experimenting with languages, operating systems and applications that establish the feasibility of distributed systems. This feasibility, in turn, means that an aggregate of arbitrarily many such interconnected and decentralized machines can render at minimum all the functions of a single centralized computer environment -- in the presence of local failures which are likely to be frequent as the number of participating machines becomes large.

Two new research groups were formed during 1982-83: The Imaginative Systems Group, headed by Professor David Gifford, is involved in the development of a community information system which uses the MIT FM station to broadcast the New York Times and other communal data to machines in several researchers homes. At these sites, special programs "filter out" and collect automatically the data of interest to each system's owner. The Theory of Distributed Systems Group, headed by Professor Nancy Lynch, is involved in the theoretical study of distributed systems. Other group changes carried out at the end of this reporting period involved the phasing out of two groups, Office Automation and Computer Systems Structure, because of the departure of their respective leaders.

Other significant events in 1982-83 were the supply by IBM of 40 personal computers to our faculty and senior researchers for familiarization purposes, and of an IBM 4341 machine for use as a simulator in our emulation facility. In addition, several members of the Laboratory worked toward Project Athena, MIT's major educational computer project that entails some $50 million of grants by Digital Equipment Corporation and IBM. Equipment Corporation and IBM.

During the past year, we were joined by Assistant Professor David Gifford, Assistant Professor F. Thomas Leighton (jointly with the Mathematics Department), Associate Professor Nancy Lynch, and Research Associate Dr. William Long. Our Laboratory consisted of 348 members -- 48 faculty and academic research staff, 30 visitors and visiting faculty, 70 professional and support staff, 100 graduate and 100 undergraduate students -- organized into 17 research groups. The academic affiliation of most of these faculty and students is with the Department of Electrical Engineering and Computer Science. Other academic units represented are Mathematics, Architecture, Humanities, Center for Policy Alternatives, Sloan School of Management, and the Research Laboratory for Electronics. Laboratory research during 1982-83 was funded by 17 governmental and industrial organizations, of which the Defense Advanced Research Projects Agency of the Department of Defense provided over half of the total research funds.

Technical results of our research in 1982-83 were disseminated through publications in technical literature, through Technical Reports (TR278-TR299), and through Technical Memoranda (TM221-TM238).

PRECEDING PAGE BLANK-NOT FILMED

COMPUTER SYSTEMS AND COMMUNICATIONS

Academic Staff

J.H. Saltzer, Group Leader	D.K. Gifford
D.D. Clark	M.V. Wilkes
F.J. Corbato	

Research Staff

L.W. Allen	M.B. Greenwald
S.T. Berlin	E.A. Martin
J.N. Chiappa	

Graduate Students

R.W. Baldwin	K. Koile
G.H. Cooper	C. Lamb
D.L. Estrin	L. Zhang
J. Frankel	

Undergraduate Students

D.A. Bridgham	L.J. Konopelski
D.C. Feldmeier	B.C. Kuszmaul
J.K.T. Genka	J.R. Lekashman
D.W. Gillies	A. Madhaven
C. Hornig	R.D. Osgood
F.S. Hsu	M.A. Pinone
F. Huettig	C.S. Rittenberg
R.W. Hyre	J.L. Romkey
E.R. Juncosa	A. Rosenstein
D.J. Karlson	J.M. Roth
L.J. Kaufman	H.J. Shinsato
F.H. Klein	S.D. Trieu
	C.M. Zeitz

Support Staff

S.C. Comfort	N. Lyall
D.J. Fagin	M.F. Webber

The work of the Computer Systems and Communications Group and the Computer Systems Structure Group this year was so closely related that a single report best describes it. The single report will be found as a separate chapter entitled Computer Systems Joint Report.

Publications

1. Clark, D.D. "Internet Protocol Implementation Guidelines," Internet Protocol Implementation Guide, Network Information Center, SRI International, Menlo Park, CA, August, 1982. Comprised of:
   Window and Acknowledgment Strategy on TCP (RFC-813)
   Names, Addresses, Ports and Routes (RFC-814)
   IP Datagram Reassembly Algorithms (RFC-815)
   Fault Isolation and Recovery (RFC-816)
   Modularity and Efficiency in Protocol Implementation (RFC-817)

2. Corbato, F.J. "Time Sharing," Encyclopedia of Computer Science, Ralston, A.(ed.), Second Edition, van Nostrand Reinhold Co., New York, 1983.

3. Corbato, F.J. "An MIT Campus Computer Network," Campus Computer Network Group Memorandum Number 1, MIT Cambridge, MA, July 1982.

4. Estrin, D.L. "Inter-organizational Networking: Stringing Wires Across *Administrative Boundaries*," Eleventh Annual Telecommunications Policy Research Conference Proceedings, Mosco, V. (ed.), Ablex Publications, Norwood, NJ, 1984.

5. Saltzer, J.H., Pogran, K.T. and Clark, D.D. "Why A Ring?" Computer Networks 7, (July 1983).

6. Saltzer, J.H., Reed, D.P. and Clark, D.D. "End-to-End Arguments in System Design," to be published in Transactions on Computer Systems.

7. Sirbu, M. and Estrin, D.L. "Cable Television Networks as an Alternative to the Local Loop," Proceedings IEEE International Conference on Communications, June 1983.

Theses Completed

1. Cooper, G.H. "An Argument for Soft Layering of Protocols," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

2. Genka, J.K.T. "A Dial Up Packet Switcher for an Internet Gateway," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

3. Hornig, C. "A Second Generation Network Interface for Multics," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

4. Hsu, F.S. "Design of a Human Interface for an Online Directory Assistance System," S.B. thesis. ! :T Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

5. Juncosa, E. "A Simple UNIX File System for the SWIFT Operating System," S.B. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

6. Klein, F.H. "Selective Dissemination Service for Users Within a Computer Net," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

7. Konopelski, L.J. "Implementing Internet Remote Login on a Personal Computer," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

8. Pinone, M.A. "A Selective Dissemination Service for Users Within a Computer Net," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

9. Rao, R.B. "The Design and Implementation of a Mail System for Interlisp-D," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1982.

10. Rittenberg, C.S. "AutoMMS: A System for Automated DEC/MMS Description File Construction," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. May 1983.

11. Roth, J.M. "Data Capture: Forms That Use Constraints," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

12. Roush, P. "Computerized Scheduling of Intramural Sports," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, July 1982.

13. Trieu, S.D. "A Transmit System, the Scheduler, for the Community Information System," S.B. thesis, MIT Department off Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

Theses in Progress

1. Feldmeier, D.C. "Performance of the Version Two LCS Ringnet Local Area Network," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1984.

2. Koile, K. "The Design and Implementation of an Online Directory Assistance System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1983.

3. Lamb, C.W. "A Screen Oriented Data Base Editor," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

4. Lekashman, J.R. "Performance Evaluation of a Packet Switching Internetwork Gateway," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1983.

5. Osgood, R.D. "Implementation of File Transfer Protocol on UNIX and IBM-PC," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1983.

Talks

1. Clark, D.D. "Internetting Local Area Networks," Conference on Local Area Military Networks, Griffis Air Force Base, New York, September 1982.

2. Clark, D.D. "Protocol Implementation and Design: Practical Considerations," SIGCOMM 83, University of Texas, Austin, TX, March 1983.

3. Saltzer, J.H. "Communications Requirements for Distributed Systems," Series of lectures, Nippon Electric Company, Tokyo, Japan, January 1983.

Committees

1. Clark, D.D., MIT Network Working Group
2. Clark, D.D., DARPA/TCP Working Group (Chairman)
3. Chiappa, J.N., DARPA/TCP Working Group
4. Corbato, F.J., CS Net Policy Support Group
5. Corbato, F.J., Advisory Committee for Health Sciences Computing Facility, Harvard School of Public Health.
6. Corbato, F.J., National Research Council: *NBS Panel for Scientific* Computing
7. Corbato, F.J., National Science Foundation: Review Panel for CS Net
8. Martin, E.A., DARPA/TCP Working Group
9. Saltzer, J.H., DoD/DDRE Security Working Group Member
10. Saltzer, J.H., Chairman, 9th ACM Symposium on Operating Systems Principles
11. Saltzer, J.H., MIT Network Working Group

COMPUTER SYSTEMS STRUCTURES

Academic Staff

D.P. Reed, Group Leader

Research Staff

M. Greenwald

Graduate Students

W. Gramlich
K. Sollins
P. Ng
J. Stamos

Undergraduate Students

R. Allen
R. Harteneck
G. Hopkins
R. Kukura
J. Leschner
J. Mracek
S. Routhier
N. Shafer
E. Siegel
S. Subramanian
T. Tran
J. Woods
C. Zarmer

Support Staff

S. Comfort

Visitors

Ø. Hvinden

The work of the Computer Systems Structures Group strongly overlapped with that of the Computer Systems and Communications Group. Consequently, the work is reported in a joint chapter entitled Computer Systems Joint Report.

Publications

1. Reed, D.P. "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, I, 1 (February, 1983), 3-23.

2. Ng, P. and Daniels, D. "Query Compilation in R*," IEEE Database Engineering, 5, 3 (September, 1982), 15-18.

3. Ng, P., Haas, L., Selinger, P., Bertino, E., Daniels, D., Lindsay, B., Lohman, G., Masunaga, Y., Mohan, C., Wilms, P. and Yost, R. "R*: A Research Project on Distributed Relational DBMS," IEEE Database Engineering, 5, 4 (December 1982), 28-32.

4. Stamos, J.W. "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," ACM Transactions on Computer Systems, conditionally accepted for publication, 1983.

Theses Completed

1. Kaufman, L. "Implementing a Distributed Debugging System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

2. Ketelboeter, V, "Forward Recovery in Distributed Systems," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1983.

3. Mracek, J, "Controlling Network Usage by Encryption-Based Protocols," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

4. Ostar, H. "An Automated Database Manual," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, December 1982.

5. Routhier, S. "An Improved Authentication Server," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

6. Topolcic, C. "Ensuring the Satisfaction of Requests to Remote Servers in Distributed Computer Systems," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1983.

7. Woods, J, "Integrating a Remote Bitmap Display in UNIX," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

8. Zarmer, C. "Implementing a Swallow Broker," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

Theses in Progress

1. Allen, R. "Validation of an Authentication Server Protocol," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

2. Harteneck, R. "A Drawing System in CLU," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

3. Gramlich, W. "Checkpoint Debugging," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1984.

4. Margolin, B. "Extension of the Multics Library System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1983.

5. Ng, P. "Library Management in the Swift Distributed System," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1985.

6. Shiroma, J. "Protocol in the Swift Operating System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

7. Sollins, K. "Name Management in a Distributed System," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1984.

8. Stamos, J. "Multi-Language Access to Persistent, External Data," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1985.

Talks

1. Reed, P. "Local Area Networks: A Research Perspective," Diebold Research Conference on Office Systems and Decision Support Systems, St. Paul, MN, July 1982.

2. Reed, P. "The Swift Distributed System Testbed," MIT-IBM Mini Conference on Advanced Personal Computers, Lenox, MA, January 1983.

COMPUTER SYSTEMS JOINT REPORT

The work reported herein was carried out by the Computer Systems and Communications Group and the Computer Systems Structure Group. The efforts were so closely related that a single report best describes it.

1. INTRODUCTION

The Computer System Structures Group and the Computer Systems and Communications Group have been working jointly on a series of related projects that are described in this joint report. The projects cover quite a wide territory:

1) The largest effort was the development of Swift, an operating system kernel with several unique features: multiple tasks in one address space with compile-time protection, heap allocation with garbage collection, and upcall-downcall organization. Swift captures in a single design many ideas developed in the last ten years on how to integrate network communications and display management with an operating system kernel.

2) SWALLOW is a unique remote file storage system that uses an append-only user interface to provide atomicity of file updates.

3) A thesis explores whether names used in computer systems can more closely resemble the way names are used by people.

4) The interconnection of networks belonging to different organizations is the subject of a new research direction that explores both the technical and policy issues that are raised.

5) The Community Information System is a new, experimental approach to dissemination of information using radio broadcast of large volumes of data and selective filters implemented by intelligent receivers.

6) A Remote Virtual Disk protocol was designed, implemented, and placed in service as part of a project to explore models of personal computing.

7) The experience of several years in designing network protocols with modularity violations was captured in a thesis on "soft layering."

8) The IBM personal computer was turned into a full scale network host attached to the Ethernet, with file transfer and remote login facilities.

9) Work continued on Internet protocol and gateway implementation, including especially an exterior gateway protocol that permits isolation of the internals of one organization network from another.

10) A ring network monitoring station was completed. Statistics can now be gathered concerning the relative effectiveness of token rings and Ethernets.

11) An online directory assistance system, DIRSYS, provides a unique interface to a telephone and mailbox directory of 20,000 names.

These eleven projects, and many sub-projects, are individually described in the sections that follow.

2. SWIFT

2.1. Milestones

The past year has seen significant progress in the Swift effort. The first eight months, roughly, were devoted to planning the implementation effort, including design of the tasking and memory management systems. Implementation started at the end of January of this year. A stand-alone system with rudimentary memory management and a preliminary implementation of the tasking system was running within a month. This implementation also included a simple driver for the console terminal and support for timers. The next month was devoted to the development of network code, including a driver for the 10 Mbit/sec token ring network developed by our group[1]and an implementation of the DoD Internet protocol. By mid-April the system was regularly sending and receiving Internet packets, and detailed performance measurements were under way. Also in April, a Remote Debugging Protocol was designed and implemented.

The detailed performance measurements carried out during April pointed out many areas where performance could be improved, particularly in the tasking system and the I/O device management. As a result, large portions of the system are being reimplemented at this time.

2.2. Lessons

Programming Language: We have gone against traditional operating system lore and implemented Swift in CLU -- a high-level, strongly typed object oriented language. This is not a radical departure; after all, Multics was written in PL/1, UNIX in C, and Pilot in Mesa. The argument against implementing an operating system in a high-level language is generally efficiency: there is a general belief that high-level languages cannot generate appropriately efficient code for the internals of an operating system. The arguments in favor of implementing an operating system in a high-level language generally include portability and ease of writing, debugging, and maintaining code -- the standard arguments for high-level languages, in general, over assembly language. Operating Systems are written in a high-level language

[1]Being sold commercially by Proteon Associates as proNET.

because the designers believe that the loss in efficiency is outweighed by the benefits of a high-level language.

There are additional arguments for implementing Swift in CLU beyond these traditional arguments. The CLU compiler enforces type safety, provides storage management, and provides and enforces useful data abstraction mechanisms.

Enforced type safety allows us to eliminate the kernel/user distinction, and to have many processes peacefully coexist in a single address space. We can trade off compile-time checking for the run-time overhead of protected areas of memory or code. This allows us to use the full features of the system even deep within the bowels of device drivers, historically a difficult place to program and debug code.

Storage management by a garbage collector allows us to freely use variable length data objects inside the kernel. A large class of operating system problems are related to allocating and freeing data objects and dangling references. We think that the system overhead of a garbage collector is a reasonable price to pay to avoid these problems.

One reason that we chose CLU is that we believed it would be easy to modify the compiler to support Swift. Though we haven't done this yet, our experience with CLU so far has been very positive.

Advantages of CLU: We were able to bring up a first system within a month of coding. Swift has been through one major rewrite already and currently consists of 4600 lines of CLU code, 1600 lines of machine language(ASM) code, and 600 lines of ASM code for the remote debugger, RDB[2].

We found the same advantages implementing Swift in CLU that application programmers have found: increased productivity, the compiler found a good number of bugs, we were able to integrate different people's code easily, and strong typing did help correct bugs.

Swift, like all operating systems, had its share of bugs, but most of them were in the ASM code. The CLU compiler detected most errors in the CLU code. After the ASM

2

Lines		Code	Comments	Blanks
	CLU	4656	1166	1587
System	ASM	1611	830	408
	Equ	1421		
RDB	ASM	633	222	207

code stabilized, almost all the bugs that were not caught by the compiler were conceptual problems with the design of Swift, and not careless errors. This was partly because of careful coding on our part, and good fortune, but it has convinced many of us that the CLU compiler is well worth the cost.

Troubles with CLU: A good portion (1600 lines of code) of the operating system is written in ASM, the assembly language interface of the CLU development system. Much of this was written in ASM for efficiency, not because it would have been impossible to write in CLU. However, most of the cost of these operations is the subroutine call necessary to invoke them. A good example of this is the word cluster -- a data abstraction we devised to deal with logical operations on 16 bit quantities. The overhead of a CLU procedure call is on the order of 20 microseconds, so to do a few simple operations on a word can take as long as 100 microseconds. If the optimizer performed in-line optimization on words, this problem would disappear.

Adding these optimizations to the compiler does not seem very difficult, and we plan to make these additions shortly.

Some characteristics of CLU plagued us throughout. A recurring problem is that of closely coupled abstractions: two data abstractions that together support an invariant. It would be useful if there were some way supported by the compiler to package the two together so that the representation of each was available to the other, but to no one else.

CLU is a high-level language, and the details of its implementation should be of no concern to the programmer. Yet, when writing an operating system, a certain amount of awareness is necessary. There are times, admittedly few, when allocations are forbidden. It is necessary to know when CLU is allocating objects from the heap. There are some expensive mechanisms under the covers that are not always obviously expensive -- programmers writing the internals of Swift must understand these.

A serious inconvenience throughout Swift was the problem of dealing with 32 bit quantities. CLU reserves a bit per longword to determine whether the longword is a reference or an integer. In application programs this rarely matters -- inside the operating system it can cause no end of trouble. There are many objects that are most easily represented by 32 bit quantities: virtual addresses, page table entries, and so on. We spent a considerable amount of effort trying to find schemes that would allow us to fit 32 bit integers into CLU, to no avail. The strategy that we adopted was to allocate objects in the heap to hold the 32 bit quantity, or store the 32 bit quantity in two integers. In cases where performance or space was a serious issue we resorted to ASM.

Deadline Scheduling: In our experience, computer operating systems often need to respond to some events in "real-time;" that is, with a guaranteed maximum latency. This is true even of systems not explicitly designed as real-time systems; it is especially important when network support is required. Thus one of the design requirements we have identified for Swift is that it be able to respond to events in real-time.

There are a number of problems involved in designing a system, especially a general-purpose system, with real-time capabilities. First, there is the issue of preemption. Maintaining a guaranteed maximum latency for response to events requires that activities in the system must be preemptible. Preemptive scheduling, of course, requires synchronization mechanisms to coordinate shared access to resources; these synchronization mechanisms may interact badly with the scheduling system employed, as will be discussed below.

In any uniprocessor multi-tasking system, the processor is a scarce shared resource, and hence its usage must be scheduled. To insure the desired low real-time latency in responding to events, most real-time systems define "priority" schemes, in which activities are ranked by order of importance and the most-important (highest-priority) activity is given the processor. The problem with such schemes is that the priorities have a global significance. The priority of a particular activity cannot be meaningfully assigned without knowing the priorities of all the competing activities in the system. In a general-purpose system like Swift, in which tasks are dynamically created and in which new programs may be run at any time, this complete knowledge is impossible and hence traditional priority schemes are not suitable.

An alternative approach to processor scheduling may be motivated by going back to our original definition of "real-time response;" namely, response with a guaranteed maximum latency. This definition suggests a natural way to schedule the processor: each activity specifies to the scheduler its maximum allowable latency; from the latencies a *deadline* for each activity is computed, and the activity with the earliest deadline is run. This approach meets the modularity goals outlined above: any activity can be designed and specified without regard to which other activities may be competing. Moreover, it has the advantage that deadlines are a very natural concept for programmers to grasp, unlike priorities (which are meaningless numbers in and of themselves).

A problem often noticed with deadline scheduling schemes lies in the specification of the deadline to be met. Most programs are composed of a variety of independently-designed modules, and it would be desirable to allow each independent module to define its own scheduling behavior and deadline. To this end we have defined the notion of a "scheduler region." Each scheduler region may independently define the deadline for its own completion. The regions may be nested, subject to the constraint that the deadlines in nested regions must be monotonically increasing (this constraint is enforced by the software). Regions both aid in the modular development of software and assist in solving the monitor interaction problem, as described in the next section.

Interactions of Deadlines and Monitors: As mentioned above, the use of a preemptive scheduling system requires the introduction of synchronization mechanisms to coordinate access to shared data; we chose the *monitor* mechanism, which integrates nicely with CLU's clusters. Other researchers have noted that such synchronization techniques can interact badly with priority scheduling systems. For example, suppose that a high-priority task has to wait to enter a monitor held by a preempted lower-priority task. If there is another runnable task with an intermediate priority, it will be run next, and will effectively delay the execution of the high-priority task.

Evidently what is needed is a way to temporarily *promote* the task holding the monitor until it can get out of the higher-priority task's way. This is particularly clean in the case of a deadline scheduler: the deadline of the higher-priority task is *propagated* to the task holding the monitor until the monitor is released. This is implemented by causing the low-priority task to enter an "implicit scheduling region" in which it will remain until it leaves the monitor in question; its deadline while it is in the implicit region will be equal to the deadline of the high-priority task waiting for the monitor.

This simple form of deadline propagation does not suffice, for it is possible that the low-priority task which holds the desired monitor is itself waiting to enter a second monitor which is presently held by a third task; and so forth. We need to propagate the high-priority task's deadline through the entire chain of waiting tasks. We can do this in a particularly simple and clever way: after promoting the low-priority task to the high-priority task's deadline, we simply wake up the low-priority task. When the low-priority task is awakened, it will again try to enter the monitor for which it is waiting; deadline propagation will again be performed, and the process will recur until a runnable task is reached.

In practice, of course, we do not expect long chains of deadline propagations to occur. It is relatively rare for a task to go blocked with a monitor locked or to attempt to enter a nested monitor, so most of the time a deadline propagation is required, the task being promoted will be runnable. In this case the overhead required by deadline propagation is minimal, and the mechanism cheaply and efficiently solves the monitor interaction problem.

High Resolution Hardware Clocks are Essential: We have relearned the lesson that many people have learned over the years: a high resolution hardware clock is essential.

Swift has no interrupt handlers. When an interrupt goes off, a task is scheduled to handle it. We expect to be able to handle interrupts from devices that have a maximum latency of 100 microseconds. This requires our scheduler to be able to handle deadlines that are specified in microseconds. We also need microsecond resolution for metering the code.

The current implementation of Swift on the VAX tries to take advantage of the hardware interval timer provided.

There are two problems with the VAX interval timer for Swift. The first is the overhead associated with updating the software clock, due both to the VAX and to Swift. The second is that the clock is jointly updated: some parts of the clock are updated by hardware, and some by software, which causes serious interlocking problems. For this reason, we would strongly prefer a better clock supported in hardware.

Remote Debugging: Experience with earlier systems has convinced us of the advantages of including debugging support even in the lowest layers of an operating system. Unfortunately, many of the facilities needed to support a reasonable symbolic debugger (such as access to a file system) are not accessible to the low layers of an operating system, especially early in operating system development. We decided to investigate an alternative approach for Swift: a *remote debugger*, in which most of the code and all of the intelligence of the debugger are moved off the machine under debug and onto a development machine. The development machine can provide all the desired supporting facilities, such as a file system, easy access to symbols and source code, logging facilities, and so forth. The machine under debug, on the other hand, contains a very small "stub" of code to carry out the debugging requests generated by the user on the development machine. The two machines are connected by a network of some description; in Swift this is the local-area network which forms the backbone of the entire distributed system. The remote debugger is known as RDB.

Several problems had to be solved in designing a remote debugging system. Although remote debugging protocols had been designed before, none was adequate for the job; so a new protocol had to be designed. In contrast to existing remote debugging protocols, RDB gives the user the capability to interrupt the execution of the program under debug at any time by sending an RDB request packet. This required tricky design in the remote debugger stub: the existing Swift network device driver had to be modified to watch for debugger packets and transfer control to the debugger at the appropriate time. The remote debugger stub then had to usurp control of the network device for the duration of the debugging session.

To date, the remote debugger has proved very useful in Swift debugging, particularly in finding problems related to synchronization and locking problems. As mentioned in the section on CLU, many of the typical problems arising in operating system implementations (such as dangling pointer problems) have been essentially eliminated by our choice of CLU as the systems programming language. Nonetheless, the remote debugger has proved worth the effort.

We envision a further use of RDB in performance measurement. In particular, we need to be able to gather statistical information on execution times and call frequencies, and then analyze this information. The analysis requires access to the symbol tables of the program being analyzed, and hence must be done on the development machine. We plan to use RDB for gathering the statistics and transporting the statistical information to the development machine for analysis.

2.3. Plans

We are still at a very early point in the development of Swift. At this point a preliminary implementation of the multi-tasking system and deadline scheduler is operational, along with a rudimentary memory management system, and the low-level support code required to run standalone on a VAX 11/750. Several device drivers are available, including a driver for the proNET token ring and an implementation of the DoD Internet protocol; the device drivers use the upcall model described above.

In the immediate future, plans call for a rewrite of much of the multi-tasking code, to reflect our improved understanding of the problem and increase performance. At the same time, we will begin using the Argus compiler being developed by the Computation Structures Group, which we expect will both improve performance and keep us on a closer track with the Argus implementation. Also in the near term, we will bring up a rudimentary, non-real-time garbage collector.

Longer-term work for the upcoming year will focus on the areas of: garbage collection; network support; file systems, including UNIX file system support and SWALLOW; and linking.

Garbage Collection: As explained above, an important goal for Swift is that it provide real-time response when needed, rendering unsuitable conventional garbage collection algorithms, which result in all computation halting while garbage collection is being performed. Several schemes have been proposed in the past for performing garbage collection in real-time; all, however, have been plagued by efficiency problems. Work is in progress on modified versions of Dijkstra's real-time garbage collection algorithm; we feel we have several promising approaches. We consider the design of a real-time garbage collector to be the most important task facing us in the next year.

Network Support: The level of network support currently provided by Swift is minimal, but the code already written is quite solid. Major tasks to be tackled in the next year are completing the implementation of the Internet protocol, including routing and error handling, and writing a version of the Transmission Control Protocol. The TCP will be a major test of the upcall model; its design will draw on previous TCP implementations done by our group for various machines.

An early goal is a version of the Remote Virtual Disk protocol designed by our group, which is presently providing remote disk access services for the Laboratory's VAXes. This is a necessary component of the file system projects described below. We also expect to soon be running an implementation of the BLINK protocol, providing remote access to bit-mapped displays and permitting work to begin on the Swift user interface.

Little effort has as yet gone into the design of the higher-level network services which will ultimately be needed. Such services as authentication and service finding will be supported in Swift through the network; much design work remains to be done in this area.

File Systems and SWALLOW: Ultimately, we expect long-term data storage in Swift to be performed by the SWALLOW distributed data storage system. SWALLOW provides an object-oriented storage system, well suited to the object orientation of CLU; moreover, it solves the problems of concurrent access to shared long-term data and of recovery after crashes.

Work to date on SWALLOW implementation has been performed on UNIX and on the XEROX Alto's, as Swift is not yet suitable for supporting SWALLOW. We anticipate that it will take us some time to learn to use SWALLOW and to modify existing applications to take advantage of its features; until then, the ability to access files on a standard UNIX file system from Swift programs would be very valuable. Accordingly, we have begun an implementation of a UNIX file system for Swift. The implementation should support the basic file system operations of creating, opening, closing, renaming, and deleting files, and the basic file operations of reading and writing blocks of data. We hope that the implementation will be useful both for use with locally-attached disks and with remote virtual disks. It should provide us with the ability to manipulate and manage long-term data well before SWALLOW becomes operational, and thus should help support the development of other pieces of the system, such as the linker.

Linking: Ultimately, Swift is intended to be useful as a general-purpose computer system. As such, it must be possible to initiate new programs and to replace existing instantiations of routines with new or updated versions. In short, we need a linking facility which permits new programs to be brought into the system, and which permits unused programs to be garbage collected. Such a linker must be able to resolve references from the newly-instantiated program to already-instantiated modules in the system; it must also provide facilities for resolving references to other not-yet-linked modules and arrange for those modules to be loaded.

Linkers may be characterized in terms of how early or late they perform the binding between symbols and addresses. There are three general categories:

1) Static linkers. The key characteristic of a static linker is that the operation of resolving free references is separated from the operation of initiating a program; at the time a program is initiated into the system, it must not have any free references. The free references are bound by an explicit linking operation, producing an executable image in which all references are bound. Replacing a module in a program requires relinking the entire program.

2) Incremental linkers. In an incremental linker, free references are resolved at the time the program is initiated. All free references must be resolved at program initiation time (although some references may be bound to "stub routines" which simply raise an error condition if they are ever called). To replace a module in a program, any current instantiations of the program are simply terminated and the program is initiated again. Note that this implies that at program initiation time, a "context" must be supplied to guide the resolution of references.

3) Dynamic linkers. A dynamic linker resolves each free reference only when the reference is actually used. When a program attempts to follow a free reference, a "dynamic linking fault" occurs, and the reference is resolved (with respect to some linking context, which must be available at run time). Replacing a module in a dynamic linking system is very similar to module replacement with an incremental linker.

A dynamic linking system is the most flexible and probably the most desirable; however, it generally requires special hardware support to run efficiently. An incremental linking system is almost as flexible and can run much more efficiently. We will attempt to implement an incremental linker for Swift.

There are a number of issues which have to be resolved before implementation of the linker can begin, including:

- The representation of programs and their static variables in memory.
- Interactions between the linker and garbage collector.
- The representation and usage of the linking contexts, which guide the linker in performing the symbol resolution.

- Details of the module replacement process, including the issues of redefining abstract data types.

3. SWALLOW DISTRIBUTED DATA STORAGE SYSTEM

During the past year, the most significant progress on SWALLOW has been the completion of a prototype broker by Craig Zarmer.

The SWALLOW broker is the software on each node of the distributed system that manages the store that belongs to that node. Such storage may be on a local disk, or remotely stored on a shared SWALLOW repository (data storage server).

Zarmer designed and built a prototype broker, running in CLU on top of the UNIX operating system. The most interesting aspect of his work was the development of algorithms for extracting data from the repository in local primary memory. The cache management algorithms must be carefully designed so that if the node crashes, with loss of the data in the cache, the system properly recovers. Thus the cache manager takes into account the concurrency control and failure recovery algorithms of the SWALLOW system.

In addition to the broker implementation, Zarmer analyzed the performance improvements due to the cache. For many applications, Zarmer's cache will significantly improve performance, as compared with a cacheless broker.

4. NAMING FACILITIES FOR FEDERATED SYSTEMS

Karen Sollins has been working on questions of how people and computers use names and how computer naming can be brought closer to human naming.

Names form the basis of communication both among humans and between humans and computers. In order to communicate with another human, the human must be able to name objects and actions in such a way that both humans understand the names. Analogously, in order to communicate with a computer, the human must be able to name operations and objects in a way meaningful to both the human and the computer. Therefore, what can be named and how is a central issue in designing a computer system useful to humans.

Sollins' work is an investigation of a naming framework for a distributed computer system, using human communication patterns to provide a set of goals for the framework. The system model is one of a *federation* of loosely coupled computers connected by a communications network. The goals for the framework based on human communication, plus the constraints presented by the federated system model, will provide the basis for the technical problems to be addressed in her thesis. *In addition, since the functions provided by this naming facility* will be different from those functions provided in past naming facilities, the thesis must address how those additional functions will be provided for the users of such a computer system.

In the past, naming facilities in computer systems have been restrictive. The space of file names was likely to be hierarchical and the name on each branch of the hierarchy might be limited in length. The space of names identifying users might be flat or hierarchical and might be limited to a small number of characters. Processes, even subprocesses, often were only nameable very awkwardly (perhaps by a number) if at all, even by a subprocess's parent. None of these has much in common with the way people name things, particularly when communicating with other people.

There are two reasons for naming entities, both having to do with communication. First, names may be used by an individual to organize and remember named entities; names provide a taxonomy. This sort of name is used by an individual or group to organize information. Second, names may be used among a group of people as the basis of communication. In order to communicate, the group must agree on the meaning of the names used. Over time, they may expand the set of names on which they agree. They will use certain protocols both to reach such an initial agreement and to expand further their basis of agreement.

The human clients of a computer system have been trained since early childhood in using a naming framework for communicating with other humans. A move toward imitation of the mechanisms used among humans would improve usability in the naming facilities provided by computer systems. The following seven observations about human use of names provide a basis for an improved computer naming facility.

1) **Communication:** *Names are the basis for communication. Therefore sets of names used by individuals should be sharable, reflecting common interests and communication patterns.*

2) **Multiplicity of names:**

- *Different people use the same name for different things.*
- *Different people use different names for the same thing.*
- *A single user uses different names for the same thing.*
- *A single user uses the same name for different things in different situations or at different times.*

3) **Locality of names:** *A person uses sets of names to reflect his or her focus of interest. A user also may use two or more sets of names to reflect a focus between or including several contexts.*

4) **Flexibility of usage of names:** *Humans use several sorts of names. For example, names are often descriptions. People use both full and partial descriptions. Humans also use generic names to label classes of objects. These generic names may be labels or descriptions. In fact, humans often use combinations of generic names and descriptive names in order to narrow the set of objects that are named.*

5) **Manifest meaning of names:** *The words used by humans for names have meanings constrained by human languages. These meanings are understood by other humans as well.*

6) **Usability of names:** *Humans are able rapidly to define or redefine names and shift contexts on the basis of conversational cues. They also have mechanisms for disambiguating names, such as querying the source of a name for further information.*

7) **Unification:** *Humans use only one naming system for all kinds of things.*

The direction in which computer systems have been moving has been toward a multiplicity of machines interconnected by networks providing a communication medium. The concerns of privacy and independence from other users have always been issues among computer administrators and users, but the nature of those concerns have changed somewhat as smaller, cheaper computers have become available. In many cases, administrators purchase such computers and put them into service in isolation. At some later time, the administrators decide to connect the computers under their management. From here, the collection may continue to grow with little control or consensus among the participants in such a "system." An *autonomous* computer is one for which all decisions are made independently of the decisions made for any other; all the activities on one computer are isolated from the activities of any other. Many administrators have pursued this option in order to escape large time-sharing systems. A *federation* is a loose coupling of computers to allow some degree of cooperation, while at the same time preserving a degree of autonomy. In a federation, there is some agreement on behavior and protocols to be utilized, but the barriers apparent in the isolated machine are still available to anyone who wants to enforce them. If the administrator or user wants to disconnect the computer from the network by simply not accepting messages, that is possible. If that computer provides a service to the participants in the network, they must understand that such a service will not always be available. On the other hand, federation provides the common ground for communication (such as agreement about protocols and services to be available) should it be desired. The loose coupling labeled federation is underlying system model of this research project.

Autonomy in the federated system limits the set of organizing structures it is possible to build. For example, sharing of information, such as collections of names, across node boundaries is restricted by the fact that the only means of communicating across node boundaries is by passing messages. The thesis will explore both the constraints from above (the clients) and the limitations from below (the federation of nodes), and will provide a naming facility conforming to those restrictions.

Briefly, the mechanisms proposed are based on two new types of objects, the *context* and the *aggregate*. A context translates names into entities. It can be given pairs of names and entities to remember and translate on demand. An aggregate is a structured set of contexts. Each aggregate has a *current context* reflecting that part of the aggregate that is being actively used by all the participants in the communication and an *environment* reflecting the private information that a participant carries to the aggregate. The current context is a single context. The environment is a collection of contexts, possibly ordered. An aggregate is an individual's view of the name resolution facility available while communicating with others.

Further work will include implementing contexts and aggregates in order to further investigate their feasibility and utility in supporting human-computer naming requirements. On the other hand, an implementation that is not a complete user environment cannot investigate fully all the issues discussed above. The thesis will consider those issues in more depth than the implementation will allow. In addition, traditionally, some naming mechanisms have provided functions that are not provided by the mechanisms of contexts and aggregates such as authentication, protection, management of other information such as time of creation or last use, and many more. The thesis will also address the problem of supporting those functions that users expect from their naming facilities.

5. INTER-ORGANIZATION NETWORKING

During the past year we continued our efforts to understand the issues raised by network interconnection across administrative boundaries; Deborah Estrin has chosen this as the subject of her doctoral research under the supervision of Jerome Saltzer. We are pursuing three related lines of investigation, each of which has both technical and non-technical components:

- What are the policy requirements for network-interconnection technology when the interconnections span administrative boundaries? How must the technology developed for *intra*-organization use be modified to satisfy these requirements, e.g., network access controls, policy filters, authentication mechanisms? How do these requirements vary as a function of the application supported over the connection, e.g., electronic mail vs. remote login.

- What are the organization implications for external interconnection? How must the connecting organizations modify existing internal policies, procedures, and configurations, all of which were established under the assumption that internal resources and facilities would be accessible to internal users only? How do these organization implications vary as a function of the technical characteristics of the connection, in particular the degree of integration with internal facilities?

- What are the public policy implications of inter-corporate networking? What is motivating such interconnection, what industry sectors are involved, and how will this new form of inter-corporate relations affect market dynamics, e.g., solidification of relationships between buyers and sellers, for example? What will be the role of public telecommunication services vs. private networks in providing the infrastructure for such interconnections?

Following an informal survey of the inter-organization networking activities that are currently underway (for example, transportation, grocery, insurance, airline, bank, pharmaceutical, university), we found that the fundamental difference between computer-communication networks that operate across administrative boundaries and more traditional inter-organization communication modes is that a user in one organization can cause some event to occur automatically within the domain of another organization, without any human intervention or auditing. Given this observation, it is useful to analyze inter-organization networks in terms of the application that is supported across the connection since the application determines the nature of event that a user in one organization can evoke in a second organization. Interconnection arrangements can be grouped into four categories of application -- electronic mail, database transaction, file transfer, and remote login. These categories differ from one another in the range of capabilities made available to external users and the degree of control over external usage available to each organization. The potential organization policy concerns intensify as the number of internal resources that the external user is given access to increases:

- Electronic mail is the most restrictive. It allows users to send and receive messages but not to extract any information from the remote system. Therefore, security concerns for the most part are limited to authentication of message originators and recipients to one another and to restricting overly burdensome volumes of undesired mail.

- Database transaction systems do allow extraction of information via querying, although typically the extent of interaction is highly restricted.

Nevertheless, due to the *active* nature of such connections, security concerns include not only authentication of the remote user, but the checking of access rights as well.

- File transfer allows a remote user to extract or insert any file such as a program, data, or document. Therefore, security concerns extend to controlling access to all stored information.
- Remote login permits access to all system resources. Therefore, security concerns extend to controlling access to all system resources.

For the most part, these security concerns are the responsibility of the internal systems' security mechanisms and not of the communication facility. But, the presence of the more diverse, external community strains what were previously adequate internal security mechanisms and policies.

One class of policy enforcement mechanisms which might be applied to insulate interconnected networks, and therefore organizations, from one another is policy filters in gateways. This is for the most part a technical fix but does require that the organizations explicitly define what their policy requirements are. The current technological basis for providing policy control between networks is almost completely non-existent. Today, whenever a packet of data arrives at the boundary between organizations it is difficult for any person or program to discover its purpose, since that purpose is buried in layers of protocols, and this packet may be only one of many that are part of a single activity (e.g., a file transfer or host-to-terminal communications stream). Present approaches fall into one of three categories, none of which provides both satisfactory function and satisfactory control:

1) Allow the packet to cross, and depend on the end points (i.e., host systems or users) to initiate only communications that meet policy constraints. This technique fails, for example, if network B finds that it can be used as a transit network between stations on network A and stations on network C. In such a case, network B gets no chance to exert any policy control.

2) Require that all protocols terminate at each gateway between networks. For every application, place a program at the gateway to act as a monitor and relay. Since the protocol is terminated, the underlying purpose of the connection is visible to the monitor, which can more easily enforce policy constraints. This approach is analogous to making a telephone call in which each party can talk only to an intermediate operator, who relays the conversation. While acceptable for some applications, delay and loss of special features cripple other applications.

3) Do not permit the connection in the first place. This approach provides conservative control, but is rather devastating from an application point of view. Given the fear of the alternatives it is probably the most widespread technique used today.

We encountered inter-organization networking activity in three arenas: industry-wide peer networks, customer-supplier arrangements, and university/research center networks. Of these three, the university/research center arena employs the most sophisticated technology and applications. We attribute the relative intensity of organization policy problems encountered in this arena to the degree of integration of each participating organization's internal facilities with its external-communication facilities. Similarly, we speculate that the absence of such integration in existing industry-wide and supplier-customer communication arenas partially accounts for the rarity with which organization policy problems have been encountered to date [see CSS Publication 4].

5.1. IBM Interconnection Project

Recently we embarked on an experimental project with IBM to study the policy requirements of interconnected organizations and to implement examples of such links. The proposed undertaking consists of two parts: implementation of a link between a MIT and an IBM local network, and investigation and study of the policy requirements that arise as a result of this interconnection.

The initial testbed for policy research will be a link between gateways attached to the MIT local area networks (largely DARPA-provided and connected to the ARPANET) and the IBM Corporate Job Network. This link will provide us with first-hand experience experimenting with policy control mechanisms that are acceptable to the interconnected parties, but minimize interference with the function and performance of the underlying data communication systems. The initial milestone of this project will be the following: electronic mail between authorized parties can be originated either within the ARPANET or the IBM network and terminate at the other network, with satisfaction as to policy control expressed by the MIT network and ARPANET operators (i.e., Defense Communications Association) and by persons responsible for asset control within IBM. Subsequent activities will include experimentation with remote login and file transfer capabilities as well as the use of information services.

The initial design of the connection is as follows: Messages destined for IBM will travel from authorized MIT users via the ARPANET and local networks to a VAX 11/750 that operates as the site of policy screening on the MIT side of the connection (Don Gillies, a UROP student, is implementing the policy-filter and mail-forwarding mechanisms for the MIT half-gateway.) After authorization, messages will be encrypted and forwarded from the *policy-VAX*, to the so-called PC-gateway, and over dial-up telephone lines to the IBM half of the gateway. The PC-gateway is an LSI-11 which forms the interface between an MIT local networks and eight dial-up ports. The IBM half of the gateway will decrypt the message files, perform any policy filtering deemed necessary, and convert the message format into one suitable for distribution over their internal network. Mail transfer from IBM to MIT will operate in a similar manner.

The encryption of messages serves to authenticate to IBM that the messages were processed by the MIT *policy-VAX*, and vice versa; in addition, encryption provides some protection from message interception. We also require a packet-level mechanism to insure that all packets arriving from IBM are forwarded to the *policy*-VAX before traveling elsewhere on the MIT networks. Jerome Saltzer, in conjunction with David Reed, Deborah Estrin, and David Clark, has specified a protocol whereby the IBM half-gateway will initiate a connection to an authentication server via the PC-gateway, before being permitted to forward packets onto the MIT local network. Once the connection has been authorized, the PC-gateway must be able to certify that subsequent packets are in fact originated by the entity that was initially authenticated. We will use a link-level protocol developed by David Reed to provide the necessary link-level authentication. This protocol encodes a *ticket* in the header of each packet (the ticket is agreed upon when the connection is first authenticated) to certify that the packet was originated by the entity that established the authenticated connection.

5.2. Network Access Control

Network Access Control is an important component of a solution to the problems of inter-enterprise communications. Network access control is our term for methods of limiting and accounting for traffic that enters a network to manage the network communication resource. That is, network access control is a way of controlling who can use a particular network, and, to some extent, controlling allocation of the network resources.

We assume that networks are interconnected by inter-enterprise gateways. Such gateways' primary job is to forward packets from one network to another. Our approach to network access control is to provide gateways with enough information to decide for each packet whether or not to forward.

An analogy is the international system of passports and visas. In this system, a person may cross a national boundary if he/she is in possession of the appropriate passports and visas. The border-crossing criterion is simple and fast to apply. Border-crossing policies, on the other hand, are implemented by individual countries through such agencies as consuls or embassies, which make a policy decision before supplying appropriate visas.

We have designed a system for network access control that resembles the visa system [see CSS Thesis 4]. Each gateway between enterprises logically combines two agencies, one for each network. When a packet arrives at a gateway, the agency for the source network will require an appropriate exit "visa" before forwarding. The agency for the destination network similarly requires an appropriate entry "visa."

Entry and exit "visas" must be difficult to forge. Our method uses a characteristic fraction computed using a reasonably secure cryptosystem such as DES (a so-called cryptographic checksum). At any point in time, the gateway knows a set of keys for use in computing such characteristic functions. To contain damage (due to lost keys, stolen "visas" etc.), keys are changed frequently in the gateway.

Corresponding to the embassies, there are Network Access Control Servers (NACS) for each network. In order to set up communication through a network, the source of messages must negotiate with each NACS for the networks its traffic must pass through. This is done dynamically. A packet entering/leaving a network with a "null visa" is forwarded to the NACS for that network by the gateway. The NACS authenticates the source of the packet using some encryption-based authentication system such as that proposed by Needham and Schroeder.[1] If its use of the network is proper, then a key to generate "visas" is sent to the source of the packet, and the packet is forwarded on.

In order that the source can properly control the path of its packets, we strongly recommend source routing [2] at the inter-enterprise connection level.

Uses for this "digital visa" scheme include control of an enterprise's information assets (asset protection) by the use of exit visas, bill source for transit network usage using entry visas, managing audit trails at the NACS, etc.

6. COMMUNITY INFORMATION SYSTEM PROJECT

Since the Community Information System project started nine months ago we have constructed software that maintains an inverted full-text database of articles from *The New York Times* and an electronic clipping service that performs selective dissemination of information. The performance and reliability of the system has now reached a point where members of our staff use the system in place of a morning paper.

The goal of the Community Information System Project is to investigate ways of using computers to help people communicate more effectively. Over the past year our emphasis has been placed on building a database of interesting information to support our on-line browsing and clipping services, keeping in mind the database requirements necessary for our extension of the service into Laboratory members' homes next year.

The Community Information System consists of many subsystems which communicate over a wide variety of communication channels. Tracing the flow of a sample piece of information should help clarify the function of the system.

Our primary information source is currently *The New York Times* news service. The news service arrives from New York at Technology Square on a standard telephone circuit. At Technology Square the signal is de-multiplexed, converted to a standard EIA signal, and made available to software on a VAX/750 UNIX system (MIT-CLS) by special-purpose hardware.

On the UNIX system a dedicated process continuously listens to the news wire and accumulates articles. The process accumulates six-level Baudot characters from the serial port dedicated to the news line, converts the Baudot characters into ASCII, watches for article boundaries, and stores each article in a separate file in the UNIX file system.

The appearance of a news article triggers several events. First, a program called the parser converts the text into our standard information item format. Information such as the author, title, priority, and subject of the article is extracted and stored in standard headers. If the article was split into several pieces for transmission the parser recombines it. Once the article is in standard format, the parser moves it to an output directory for the next step of processing. The parser also uses the information it extracts from articles to maintain a synopsis database which is used by the on-line browsing program.

Once the article has been converted to a standard format, it is included in a full-text inverted database of on-line information items. An online tool called "browse" allows users to access articles by specifying a desired article's category, type, or by specifying free text keywords that appear in the ar' .le.

In addition to maintaining an on-line database, users can send mail to "Clipping@MIT-CLS" to specify a standing query or "filter." After an article is processed by the parser it is examined by the clipping service. The article is mailed to any user who has submitted a filter that matches the article. Filters are boolean combinations of words and phrases that can be applied to the priority, author, and text fields of an article.

The database system and clipping service we have built are general purpose, and are not limited to processing information from *The New York Times*. The design of the system is intended to make the addition of new sources of information as easy as possible. For example, we are currently finishing software that will allow users to make database submissions via ARPANET mail. Once this software is complete, ARPANET bulletin boards will be included in our database.

In addition to central site services, we are making our data base available to geographically dispersed computers. This is accomplished by broadcasting information on a low-cost digital packet radio system. Remote computers use the broadcast information to update their local databases according to the interests of their owners. The scheduler for the broadcast channel is complete, as is the engineering work to start digital broadcasting this summer. By next year the software for the remote computers will be complete. The software will keep remote databases up to date and display information according to priorities set by its operator.

An important lesson that we have learned this year is that text need not be indexed by hand to be useful. Full-text indexing of articles proved to be very effective in allowing users to select relevant sets of articles from the database. Part of our continuing interest lies in human engineering the system's user interface to permit non-professional users to use it as an everyday tool.

7. MAKING THE VAX LOOK LIKE A PERSONAL COMPUTER

We have purchased a number of VAX 11/750s from DEC. These are meant to be used for research in single user machines.

We firmly believe that personal computers as powerful and as large as the VAX will be available to the consumer within five years. These computers will fit on a desk-top and will be reasonably priced. We want to determine interesting ways to use these computers *now*, so that when the technology arrives, we will be prepared.

Simply using the VAX as a single-person computer does not make it a personal computer. Personal computers possess certain characteristics. A personal computer is always accessible, operates for you continuously, and is configured to your personal taste. It does not have to be portable, but it should not be difficult to move it. The VAXes do not have these features.

The VAX is large, hot, and noisy, and is therefore not suitable for your office or your home. It is not accessible in the same way that *personal* computers are. It is possible to attach a line from your VAX to your office and maintain the illusion that the VAX is in your office, but it is not terribly easy to move the VAX to another office. Several drawbacks come about because there are fewer VAXes available than people who want to use them as personal computers. This means we must take turns using a VAX as our personal machine. Sharing personal computers presents several problems. Sharing prohibits continuous operation on your behalf. It is not polite to simulate a circuit for several days if people are waiting to use the machine. If more than one person uses your "personal" computer, whose taste is it tailored to? If we are able to configure a computer according to people's tastes, does this then mean that they can only use the single computer that is theirs, even if several other VAXes are sitting idle?

We have performed research aimed at making the VAXes act like personal computers. This entailed maintaining the illusion that each person had a VAX in his or her office that belonged to them *personally*. We accomplished this by attaching the VAXes to a local area network[3]. Two research projects dealt with making the VAXes your "personal" computer. BLINK allowed you to attach a bit-mapped display with an input device to the VAX. This display sat in your office, and was connected to some network. RVD allowed you to attach your "virtual disk drive" to your computer. The virtual disk drive is really only a segment of a large disk that everyone shares.

7.1. RVD

Our ideas about personal computers were influenced by our use of XEROX Alto's. Each Alto was identical. Each user had their own disk(s) that they inserted into the Alto's disk drive when they booted the machine. We tried to do something similar with the VAXes. The result was RVD - the Remote Virtual Disk protocol.

The goals of the RVD project were to find a way to allow you to approach any available VAX, "spin up" your disk, and have a personalized environment. Since the VAXes were located far from our offices, and RK07 disk packs were expensive, allocating an RK07 pack to everyone was unacceptable. RVD provides each machine with many "virtual" drives in which a user can "spin up" any of his disks.

Other advantages of remote disks are the ability to use machines that have no disk drives, the ability to obtain economy of scale by purchasing secondary storage in large chunks rather than an RK07 at a time, and to share common code, rather than duplicate it on everyone's disk[4].

Why did we implement RVD as opposed to a file server? RVD provides more

[3]The Proteon proNET (also known as the Version 2 ring). The proNET is a 10 Mbit/sec ring network.

[4]The savings can be enormous. At MIT, a complete UNIX (man, sources, lisp, pascal compiler, CLU system, and so on) is larger than an RK07. A scaled down version still takes a substantial portion of the disk. Storing most of UNIX on RVD disks allows the full use of UNIX, and allows each of the 22 disks to be used for useful storage.

flexibility. A file server imposes the clumsy model of files on its clients. Some of our systems have no notion of files. In general the concept of adding on some number of disk drives seemed much cleaner and natural than the complicated idea of sharing files, and retrieving them from some common source. Adding a drive to page off is understandable in a virtual disk system. Mounting a filesystem on top of a disk drive seems very straightforward. This allows you to slip in a shared file system underneath the operating system transparently. Once you have spun up a disk, it is attached to your computer. You do not have to negotiate with the remote server for each file access.

RVD consists of two halves -- the RVD server and the RVD client. The RVD server manages the disks, and responds to the clients requests over the net. The current server implementation is running as a user program on a VAX 11/750 running UNIX. It has three RA81's attached to it. The protocol was designed to minimize the computational overhead at the server so that it should not be a bottleneck. The RVD client is inside the UNIX kernel. We wrote a device driver for virtual disks, and it fits neatly under the UNIX operating system.

RVD has been in service for about eight months. Until this month the server was running as a user program on a time-sharing system with a single RM80 used for RVD. From the client machine reads are comparable to an RK07, while writes take about twice as long. The current version of the server was a quick and dirty hack, written originally to run on a PDP 11/45. It was meant to test the client code. When the server is rewritten we expect to see write times comparable to read times.

Because of the limited disk space and the slow writes, we have limited RVD to be used for shared read-only disks, and for file system backup. A full UNIX can take as much as 75% of the disk space on an RK07. Most of UNIX is now run off of virtual disks. RVD has also been very useful for tape backup. Without RVD we would be required to shut our VAX off, and physically carry our RK pack to a machine that has both a tape drive and an RK07 drive. With RVD we just spin up a backup disk on both machines, and copy the appropriate dump to tape, while our system is running.

We have just brought up our RA81 drives. This has added more than a Gigabyte of storage to RVD. The writable disks have just been allocated, so we should probably see an increase in usage of RVD. Currently, the RVD server receives about 500K packets every two days, and sends about 1M packets in the same interval.

8. AN ARGUMENT FOR SOFT LAYERING OF PROTOCOLS

During the year Geoffrey Cooper completed a Master's thesis concerning protocol layering. There are two ways of looking at what the thesis accomplishes. From one point of view, the thesis begins with the fact of layered protocols, analyses them, finds them lacking, and suggests a "better way" to write protocols. From a different point of view, the thesis develops the need for layered protocols, examines their advantages, and suggests how layering may be maintained in a protocol design without undue cost to the efficiency of the protocol implementation.

The thesis concentrates on one particular situation, that of a layered protocol architecture which implements reliable communications between cooperating application-level entities. In this context one sees that the maintenance of a layered structure in the protocol implementation could cause it to be so inefficient as to be unusable. After a good deal of analysis, an extension is introduced to protocol layering which provided a mechanism whereby the shortcomings of the layered structure can be fixed.

The thesis begins with a development of the concept of protocol layering, and an outline of the advantages of the scheme. This discussion notes that because there are typically many different network entities inside of a computer system, but only one (or perhaps two) hardware interfaces to the network, it is a requirement of the network software in the system that it allow all of these entities to share the network hardware. Further, because the task of implementing all the network applications in a host is a major one, there is a strong desire to modularize the structure of the network software in a computer system in such a way as to make it possible for the different network applications to share, at least, part of the code that implements them.

Protocol layering provides an elegant means of satisfying these two criteria in a single mechanism. In a layered protocol architecture, each layer of the architecture provides to the layers above it a more sophisticated set of services than is provided by the network hardware. The nature of the "refined" service that is provided by each layer is such that the service fits into the requirements of many of the network applications. Some applications will wish to make use of this refined service directly, while others will make use of it indirectly through the device of added layers of protocol (each of which refines the service further). Protocol multiplexing can also be provided in each layer of the protocol architecture, to allow applications to use the layer's services directly without letting them interfere with transport layers which wish to further refine the layer's services. As a side effect of the introduction of protocol multiplexing, the requirement of being able to share the network software among many network entities is also met.

Protocol layering is, then, a powerful technique for achieving modularity of the design and implementation of network software. It has been central to the discussion of the preceding chapters that the advantages presented by protocol layering are sufficient that it would not serve to abandon a layered structure entirely.

Still, protocol layering is not without severe shortcomings. Because of the uncertainty which is inherent in all network communications, a network entity must always maintain a *death timer*, which provides it with a mechanism that it can use to avoid waiting forever for a cooperating remote entity which has failed. In a layered protocol implementation, the same problem occurs at every level of protocol, so that it is necessary for the implementation of each layered entity to provide its own death timer. Since, at a given time, the cooperation of all the layers in use is required for any useful communications to take place, it clear that all but the shortest of these death timers are really unnecessary. All but one of the death timers in a layered structure is thus a parasitic side effect of the introduction of protocol layering.

It is also common practice to set shorter *optimization timers* for the purpose of providing different transmission characteristics than are available in the lower layers of protocol. Since lower layers may piggyback messages with those of higher layers, an optimization timer at a lower layer which is longer than one at a higher layer is redundant. Thus, protocol layering encourages redundancy of optimization timers as well as death timers. The characteristic of layered protocols that caused their implementations to set many timers, most of which are useless at any given instant, was entitled the "timer problem."

A more severe problem than the timer problem is also investigated in the thesis. Entitled the *asynchrony problem*, it stems from the need for network entities to reliably coordinate state information with their cooperating remote entities. In a layered context, this coordination of state occurs independently at each layer of protocol, because of the perceived inability of layered protocols to coordinate this activity without violating their modularity. The asynchrony problem results in an increase in the number of packets sent over the network to perform a given function at the highest level of protocol. This increase is (in the worst case) exponential in the number of layers in the protocol architecture. Since many of the costs associated with sending and receiving packets are independent of their size, an exponential increase in the number of packets sent and received can be expected to result in a roughly exponential *decrease* in the relative throughput as seen at the application level protocol. It is thus a requirement of any protocol implementation that was to provide a useful service that it avoid the asynchrony problem.

There exist protocol implementations that *do* provide a useful service. The thesis examines some of the techniques that are used by these protocol implementations to avoid the .synchrony problem. These range from the total abandonment of a layered structure to a series of predictive "tricks" which work well for some higher level protocols, some of the time. The inherent harm in these techniques is that each is entirely independent of the protocol specification. Thus, to implement a usable version of a protocol, it ceases to be sufficient to simply implement its specification as written. If a protocol specification does not say how to implement the protocol, then its value is considerably diminished. Furthermore, the "tricks" needed to implement a protocol efficiently are generally not codified, and are often specific to particular protocols or operating systems.

The existing solutions to the asynchrony problem make clear the attractiveness of any solution to it that works *within* the context of a protocol layering, and is integrated into the protocol specification. The major effort of the thesis was to develop such a technique, which is called *soft layering*.

In a soft layered protocol, the protocol specification is augmented to include a "usage model" for the protocol: a model of the way in which the protocol expects higher level protocols to use it. Higher level protocols which conform to the usage model may expect to receive an efficient service from the protocol being specified. Other higher level protocols will still be able to make use of the service defined in the protocol specification, but the service provided to them will not be efficient. Soft layering provides a mechanism whereby the meaning of protocol efficiency -- which is *always* a part of the protocol implementation -- may be formalized in the protocol's specification. This ensures that all implementations of the protocol provide the same service from the point of view of both correctness and efficiency.

The analyses of protocols that were performed in the thesis led to another of its contributions. The thesis develops a remarkably succinct and useful terminology for analyzing network entities: "happiness terminology." A network entity is said to be *happy* if it has received confirmation from its cooperating peer entity, indicating that every action requested of the peer has been completed (successfully or otherwise). It is *unhappy* if this is not the case: if there is some action which has been requested of the cooperating peer for which no confirmation has been received.

It is our belief that the concept of "happiness" is generally useful in the process of designing and implementing protocols, in a manner analogous to the way in which data abstraction is useful in the process of designing and implementing other kinds of software. For example, the question of how a protocol entity is made happy or unhappy is analogous to the maintenance of a rep invariant in an abstract data type.

9. IBM PC NETWORK SOFTWARE PROGRESS

This project started one and one half years ago with the goal of making a personal computer act as a full-scale network host. The first step was to implement a file transfer program based on the Department of Defense InterNet family of protocols on an IBM Personal Computer. The second major application was the remote login program, Telnet, based on TCP/IP, and on which much of the work of the past year has been done. Initially, the plan was to run the network protocols over a serial line to a gateway to the high speed networks. More recently, direct attachment to local area networks has been added.

9.1. Internet Implementation I

Last year's progress report described a very efficient file transfer package, TFTP, that achieved its effectiveness partly by being very non-modular. This year's goal was to insert modularity without losing effectiveness, so that a common internet layer could be used for both file transfer and remote login. The first step was to port the internet and UDP code from a UNIX implementation done by Larry Allen. We tried to preserve the software interface for the routines, but a major goal was also to prevent unnecessary copying of packet buffers between layers. Since the interface driver and the user program run in the same address space in our implementation on the PC, we were able to get away with copying data just twice: once into or out of the packet buffer for the user program, and once to or from the device.

The UDP- based name user code from the UNIX implementation was also ported with UDP. It resolves textual host names by polling known name servers over the network, and is integrated with all of the user packages.

We also needed a network interface driver. Anticipating a need for several different network drivers, we modified the terminal emulator's serial line driver, gutting it and adding C code to deal with the link level protocol that we use over the serial line. We also required a serial line driver for the gateway that we used, so that packets generated by the PC could be forwarded to another network. We used Noel Chiappa's C-Gateway. David Bridgham wrote this driver, though its development was hindered by the rapidly changing hardware and software substrate (the C-Gateway was still under development at the time).

Louis Konopelski ported Larry Allen's TCP and Liza Martin's Telnet to the PC. This effort was simplified by the fact that our internet layer kept almost the same interface characteristics as the one written to work with TCP. The TCP uses a small non-preemptive multi-tasking package which allows it to be quite responsive to the asynchronous nature of the network. It also uses some of David Clark's ideas about upcalls.

We also decided that it would be good to have a TFTP which used the same internet library as the TCP, so we ported that TFTP from UNIX to the PC also, at negligible performance loss over the old one.

9.2. Internet Implementation II

We decided to modify the internet implementation to take advantage of the conditions on the PC under which it was running. On the UNIX system, it ran in a user process and communicated with the kernel via system calls. Here, with all the network code in a single address space and with our tasking system, we could do better than that, and have a consistent structure throughout the system based on tasks and upcalls.

The new implementation has a task associated with each network device. When a packet is received, the interrupt handling code enqueues the packet and wakes up the task. Later, when the task runs, it removes the received packet from the queue, does the processing of the packet that needs to be done at this level (protocol de-multiplexing) and calls internet with the packet if it is an internet packet. Then internet does its processing and calls TCP, or UDP, or ICMP, or GGP in turn with the packet.

When a layer wishes to send a packet, it fills in the parts of the packet that it wants to fill in and then calls the layer below it with the packet. Finally, the internet layer routes the packet, looks up the address of the routine to physically transmit the packet, and calls it.

This modification required an almost complete rewrite of the internet and UDP layers, as well as the introduction of ICMP and GGP code. (The only GGP function supported is an Echo server). TCP was quickly modified to utilize the new structure, and it worked well.

January saw the release of a new network interface, the 3COM 10Mb Ethernet interface. We had to develop a driver for it for the PC, done by John Romkey, which slipped in modularly in place of the serial line driver, and also one for the C-Gateway, ported from BBN code by David Bridgham. In addition, we needed some way to translate from internet addresses (which are 32 bits long) to Ethernet addresses (which are 48 bits long). To do this, we chose the Internet standard Address Resolution Protocol, which also required implementation on the C-Gateway.

The new structure of the system proved itself the first time we tried an Ethernet Telnet. After resolving some differences in the implementation of the Address Resolution Protocol between the PC and the gateway (this was the first time they had ever spoken to one another), and one bug fix in the PC code, we had a working Ethernet Telnet.

At this point, further development was done on the Ethernet driver and much time was spent refining TCP and Telnet. We also wanted to bring TFTP up on the Ethernet, but the implementation which we were using would not port easily to the new internet, nor could it easily utilize a new driver. A new TFTP was then written from scratch. A TFTP to floppy disk typically has transfer rates around 15 Kbit/sec; TFTP's which discard the incoming file have run as high as 98 Kbit/sec. These TFTP's were with a PDP 11/45 running UNIX, and were done via the C-Gateway.

As we used the programs which we developed more and thought about the possibilities of having them run at other sites than MIT, we encountered several issues which caused us to build a customizer. The issues included

- having a single program run at different speeds on the serial line
- determining the PC's internet address on an Ethernet
- determining the addresses of name and time servers
- initial values in the internet to Ethernet address translation cache
- setting up personal attributes to Telnet (such as whether the back arrow key is delete or backspace)

There is a data structure in a well-known place in every program that contains initial values for these attributes and others such as what the program is, the version number of the program, and when it was last customized. Every program uses the same data structure, and the customizer simply allows editing of this structure through a menu-oriented user interface, as well as duplication of the structure of another program. These changes previously required recompilation of the program.

A number of other programs were also developed. They include TCP *whois*, which queries a remote site for information about one of its users; *ping*, which sends out ICMP echo requests; *setclock*, which queries a set of time servers and sets the PC's clock based on the results; *hostname*, which resolves textual host names into numeric addresses and prints the addresses and the names of the servers which responded to the request; and *cookie*, a program which prints a "quote of the day" after having fetched the quote from a *cookie server.*

9.3. The Terminal Emulator

The terminal emulator was originally developed by David Bridgham to allow PC's to be used as terminals during program development. It emulated a DEC VT52 at the time. It was later upgraded to emulate a Heath H19, with the exception of ANSI mode and certain things such as keyboard locking which would be impractical on the PC.

During the development of Telnet, we realized that it would be much more useful to have the PC appear to be a smart terminal such as an H19, rather than as a dumb terminal which could do no cursor or screen manipulation. We decided to slip the same low level terminal emulator code in the standard I/O library terminal output code. This involved breaking the emulator up into two distinct parts, which handled the user interface, serial line, and actual emulation.

Experience with Telnet later showed that at times, Telnet could receive data for the screen faster than the emulator could handle it; the emulator became a bottleneck. This problem was rectified by having the hardware do the scrolling instead of the processor, as was the case before. With the improved emulator, Ethernet Telnet often performs better than a 9600 baud line wired directly to a machine.

We also found it necessary to add handling of some ANSI mode features since EMACS on one of the machines that many people around the lab use utilizes ANSI mode operations.

9.4. Remote Logging Protocol

To aid in debugging of a variety of programs, we implemented the remote logging protocol described by Dr. David Clark. Use of this service allows us to monitor machines that are in service, discover the reasons for failure of machines, and see when obscure conditions which should never occur do occur. A server for the logging protocol was done for VAX UNIX in CLU by Mark Rosenstein. User logging code was written for the C-Gateway, the IBM PC, the MIT TFTP Dover Spooler, and the Network Monitoring Station by David Bridgham, John Romkey, Geoffrey Cooper, and David Feldmeier.

10. INTERNET PROTOCOL WORK

10.1. Protocol Performance Improvements

Work continued this year on testing techniques for achieving higher performance from the Internet family of protocols on different types of computers and different operating systems.

10.2. UNIX Kernel Network Support

Many of the Internet protocols were implemented on our PDP 11/45 running Version 6 UNIX. Since the kernel in this system has a small address space and a poor debugging environment, only the the most basic networking functions were included in it. In the kernel are modules to drive a proNET ring device, to transmit and receive Internet packets, to de-multiplex incoming packets, including UDP and TCP packets, to reassemble Internet fragments, to maintain a cache of Internet hosts and their best first hop gateways, and to route a packet to its appropriate first hop on the local net.

10.3. Application Processes

Outside the UNIX kernel are network application processes to handle remote login, file transfer, mail transfer, and network diagnostic, error and routing reports. The user and server Telnets deserve special mention since they run on TCP which is the most complex of the internet protocols.

Server TCP/Telnet: Server Telnet, the remote login protocol, runs in the same process with its TCP and IP layers; it supports one Telnet connection. Multiple server Telnet processes run simultaneously when several remote logins are being supported. Collapsing server Telnet into the same process with its TCP and IP has several performance benefits - such as eliminating interprocess communication and data copying, and decreasing the number of processes scheduled. It also allows TCP to query Telnet about any data it might want to send out with TCP connection maintenance information; this reduces the number of packets transmitted on a connection. The jobs of the various layers can be performed when most appropriate rather than when each protocol layer is "scheduled." This implementation of TCP supports most features of the protocol and includes code to prevent "silly window syndrome."

User TCP/Telnet: Network protocols are often specified with a large amount of internal asynchrony; this greatly complicates their implementation. This is particularly true in systems like UNIX, in which processes cannot share memory or communicate cheaply. The result is that protocol implementations often use special-purpose multiplexing to simulate asynchronous activities; this muddies the structure, making understanding and modifying the code difficult.

We explored a different approach with our implementation of user Telnet. We designed a small subsystem permitting multiple asynchronous activities ("tasks"), each with its own stack and machine state, to run within the context of a single UNIX process. Tasks can be scheduled in response to events external to the process, such as the arrival of a packet from the network, and by other tasks within the same process. A non-preemptive scheduling algorithm is employed to avoid coordination problems in accessing shared data.

The user Telnet includes a small implementation of the Transmission Control Protocol. This implementation was actually a translation of a TCP written in BCPL by David Clark for the XEROX Alto minicomputer. The TCP implementation uses three tasks, of which two are contained in the TCP module and one deals with timer management. One TCP task handles input packets; it is awakened by receipt of a signal from the network driver indicating that an input packet is available. The other TCP task handles packet transmission; it is awakened by the TCP receiver task (to send acknowledgments), by the user of the TCP (to send data), and by timer expiration (to perform retransmissions). The TCP tasks communicate with each other by sharing state variables, while communication with other protocol layers is by means of subroutine calls. A fourth task runs the actual Telnet implementation.

Trivial File Transfer: Due to the size and complexity of the full ARPANET File Transfer Protocol, we have not yet completed an FTP implementation; instead, we chose to implement the Trivial File Transfer Protocol (TFTP), a simple file transfer protocol built directly on datagrams rather than on a stream connection. This simple protocol has proven very durable and useful in the past; in addition to file transfer service, it is used for remote printer access, network bootloading, and has been used for mail transport.

It should be noted that, despite its relative simplicity, TFTP exhibits in microcosm most of the implementation difficulties found in network software in general. It is significant to note that the current implementation of TFTP on our UNIX achieves roughly three times the throughput of our previous UNIX implementation while occupying roughly half the space; this suggests the effects of the learning curve in network protocol implementations.

Simple Mail Transfer: The simple tasking TCP designed for user Telnet also forms the basis of the Simple Mail Transport Protocol implementation. Some extensions were required to add "server" functionality, but the modifications were small.

10.4. Results

We have seen as many as eight active remote login sessions at once with reasonable response times. Some performance measurements have shown the following results. Round trip time for an Internet Control Message Protocol time stamp packet sent from the PDP 11/45 to itself took between 30 and 40 milliseconds; this required two packet transmissions and two receptions. The maximum TFTP transfer rate that has been observed was 133 Kbits/sec between our machine and a VAX on the same ring net. The TCP used in user Telnet has been observed to sink data in a memory-to-memory transfer from a VAX at 215 Kbits/sec. The server Telnet's TCP has been seen to send data in a memory-to-memory transfer to an Alto on an Ethernet at 300 Kbits/sec; the gateway between the proNET and the Ethernet was an LSI 11/03 running our C-Gateway code.

10.5. Gateway Implementation

Exterior Gateway Protocol: MIT has been participating in the development and implementation of a new protocol to be used to pass routing information between systems of autonomous gateways. The protocol is called exterior gateway protocol (EGP); its purpose is to provide a more controlled method of passing routing information between gateways who may or may not trust each other.

A loose definition of autonomous gateways is that they belong to the same administrative organization, such as MIT, and are fairly homogeneous. The gateways within an autonomous system will use their own conventions for passing routing and up/down information among themselves, and will use EGP to pass routing information between themselves and the outside world.

Summary of C Gateway Progress: Copies of the C-Gateway were sent to Stanford and BRL. These installations now seem to be running quite reliably.

A lot of effort was put into making the C-Gateway more robust; about two dozen bugs were isolated and fixed, and logging to a VAX at startup time was added to monitor gateway crashes. It appears that the MIT-GW crashes about 5 times a day. Most of these crashes are soft crashes in that the machine just restarts without needing to reboot itself.

Better routing mechanisms were added or worked on such as a default gateway for packets destined to a net to which we have no route, ICMP redirects, and EGP.

So much work was done on the C-Gateway this year that the easiest way to describe it is with excerpts from monthly progress reports.

1) August and September 1982

> Much has happened on the C-Gateway over the last several months. It is now in full operational service in the main MIT. ARPANET gateway, and will shortly go into service at Stanford in the ARPANET gateway there and in several local network gateways at MIT
>
> Substantial work has been done on the internal structure to speed it up even further, and a fast and simple general tasking mechanism with priorities has been implemented to allow finer control over internal work scheduling. Extensive analysis of operation in high-throughput applications internal to MIT is proceeding; some preliminary results have already resulted in improvements. In one test at MIT a while back, an LSI-11 was able to maintain a data rate of 1/2 Mbit/sec from a proNET to a 3MBit Ethernet; this is quite good considering the slowness of the LSI-11 and the fact that each packet must be byte-swapped before being sent out on the Ethernet. Indications are that a faster processor with two DMA interfaces could sustain data rates in the several megabit range. Some initial investigation of data flow and flow control inside the gateway has been done, and further modifications to allow more control in this area are planned.
>
> The code to handle PUP was written; it is now relatively complete, providing full simple gateway service, which is to say complete routing and ECHO but not name resolution or boot servers. The CHAOS protocol code was redone, and is now considered complete at the same level of simplicity (CORUT, STATUS and PULSAR). Work to expand the IP implementation to a full IP (including ICMP, GGP, class A/B/C

support, etc.) has started; this was delayed until the previous two protocols were done to allow the experience gained there to be included. The ARPANET driver has been cleaned up, and plans to expand it to include per-link flow control are complete. Finally, an extensive audit of the code has been made to remove places where, in the initial rush to get the code up, bughalts were placed on unlikely code branches.

2) October 1982

As a result of analysis done last month, some major changes were made to the structure of the gateway software to allow better internal flow control as well as packet buffer reclamation. All packets are now kept on explicit queues instead of implicit ones (i.e., system message queues, etc.), and internal flow control code was installed. No code for external flow control (Source Quench ICMP packets) has been installed because the algorithms are still under consideration, but all the necessary hooks exist. Memory allocation was speeded up by keeping an internal list of free buffers instead of using the MOS memory allocation scheme. A smarter buffer allocation scheme (using loose pools and minimum reservation strategies) was installed.

The code has been deployed in additional sites at MIT, including the main gateway between the high-speed local networks at Tech Square, where it performs adequately. With the addition of a MOS device driver for the ACC ARPANET interface, obtained from SRI, a C-Gateway was configured for Stanford University to interface between the ARPANET and the collection of Ethernets there; this came up with almost no problems and packets were exchanged between a VAX on an Ethernet at Stanford and a VAX on a proNET at MIT.

3) December 1982

The C-Gateway code was packaged and shipped to the Army's Ballistic Research Lab in Maryland for installation in a gateway there. BRL will be writing additional device drivers and handlers as needed.

4) January 1983

Efforts are being made to improve the reliability of the C-Gateway and to collect more information about crashes. Software bughalts have been changed to save information about the crash and then to restart immediately. Also when the gateway starts up, it now sends a log packet to a log server on a VAX on the proNET ring; this log packet includes the message that was stored away by the last crash. Currently log packets are not always getting to the lo^ server; this may be because when the gateway starts up, it reenters the proNET ring causing some perturbation in the ring.

Code to respond to ICMP pings was added to the gateway. Code to generate ICMP redirects when needed was also added. Due to fears about possible adverse effects on ARPANET hosts, the gateways presently send redirects only to hosts on the LCSnet.

5) February 1983

A driver for the Interlan 10Mb Ethernet interface was written and installed in the IBM PC gateway. It uses David Plummer's Address Resolution Protocol (RFC 826) to translate 32 bit internet addresses into 48 bit Ethernet addresses.

Slightly improved routing code was added to the gateways. Packets destined for nets to which a gateway has no route are now forwarded to a default gateway. This is a temporary solution to the Class B/C net problem. As a result, hosts on the LCSnet can access all Class B and C networks.

The Version 1 ring network code and interfaces were removed from our gateways which means that this network has been decommissioned.

6) March 1983

The strategy for resynchronizing with the IMP when the IMP goes down has been changed to be similar to the strategy used by the BBN gateways. The old strategy was not robust in the cases where the gateway thought the IMP went down, but in fact the IMP was still up.

Ron Natelie at BRL-BMD is running an old version of the C-GW code that he has modified somewhat. He found a bug in the ACC driver that causes the gateway to think the IMP went down. His fix to this bug is included in the current ACC driver running on MIT-TGW.

Most of the code to implement EGP was written this month. As one would expect with an early implementation, the implementing process has turned up suggestions for some fairly minor modifications to the protocol.

7) April 1983

Several releases of the C-Gateway were sent to Stanford University this month. Some bugs were isolated and fixed; currently this gateway is providing a barely acceptable level of service. Debugging continues.

The BRL-Gateway version of the C-Gateway is running as a regular service now.

8) May 1983

Efforts on the part of Jeff Mogul and Bob Baldwin to debug the version of the C-GW running at Stanford turned up various packet length bounds problems. Jeff noticed that better packet length checks were needed in the Ethernet device driver and Ethernet network handler. When these checks were installed, the Stanford gateway became substantially more reliable. It now crashes about once every two days.

Earlier in the month Jeff and Bob figured out that a problem that looked like an IMP synchronization bug was in fact due to a feature of the ACC DMA interface to the IMP. The ACC device transfers two garbage bytes into memory at the end of the received packet. The extra bytes cause an input overrun when the gateway receives a maximum-sized packet. The device driver did not check for the overrun, so the garbage bytes would appear at the beginning of the next packet, causing it to be discarded due to bad ARPANET header format.

Our implementation of EGP was tested to itself over the MIT test gateway's two interfaces (ARPANET and proNET) and to itself via an Echo server. It was also tested to DCN6 and DCN1. These tests have shaken out a few bugs; EGP seems to run quite robustly in the test gateway now.

MIT took receipt of a Bridge Communications 68000 based computer this month. It will be used to develop experimental gateways.

11. NETWORK MONITORING STATION

In the field of local area networks, two types of networks dominate: an Ethernet type bus network and more recently the token ring network. Much of the current interest in rings seems to be due to the pending announcement by IBM of its token ring network. Although there is much debate over the relative merit of both types of networks there exists little information on the performance of rings. John F. Shoch and Jon A. Hupp of the XEROX Palo Alto Research Center produced a preliminary report on Ethernet performance containing such things as number of packet errors, performance under high load, stability and fairness. Unfortunately, no such comparable document exists for a token ring network. The purpose of the Network Monitoring Station is to collect statistics on the LCS proNET ring.

The Network Monitoring Station currently consists of a PDP 11/20 mini-computer with a network card for the proNET ring and some specialized hardware. The network cards for the proNET ring have two parts. The first is a Control Card which interfaces to the network on one side and has a standardize interface on the other. The second board, the Host Specific board (HSB), has this same standardized interface on one side and a host specific interface on the other. The special hardware is placed between these two cards. The specialized hardware has three major components. The first is a HSB emulator that interfaces with the Control Card. The second is part is a Control Card emulator that interfaces to the Host Specific Board. The third part is a 32 bit, 40 microsecond resolution crystal clock.

The HSB emulator is the simplest. It always appears to be an HSB that is ready to accept a packet. No matter what the state of the true HSB, it will always receive a packet from the net. The Control Card itself is set in the "match all" mode which simply means that any packet that comes by will be received no matter what its address. The HSB emulator counts the number of incoming bytes and also keeps track of error signals received from the Control Card.

The Control Card emulator acts as a Control Card that only receives 17 byte packets (actually the first 17 bytes of an incoming message). By only working with the first 17 bytes of a packet, the Monitoring Station obtains the necessary information from the IP protocol and allows the HSB the maximum amount of time to reset for the next packet. The Control Card emulator keeps track of whether or not the HSB has received the incoming packet.

The clock is used for timing events on the ring and for maintaining the current time of day. Other things that are monitored include the times that the ring crashes and reinitializes, bad format packets, and hopefully soon, packets that are refused (not received by the addressee). The specialized board is memory mapped into the PDP 11/20 and also has an interrupt mechanism for fast retrieval of packet information.

The software running on the PDP 11/20 does some data compression, simple data accumulation and analysis. The only two items on the bus that can interrupt are the HSB and the specialized hardware card. These interrupts are fast (~20 microseconds) and simply place data from on board registers into a circular buffer. From here, the information in these buffers is analyzed when there is time. The software accumulates statistics such as number of packets and percentage of netload over the previous day, hour, minute and second. Also displayed is the time since last network crash (loss of token). Both network errors and monitoring station errors are accumulated.

Because the Monitoring Station has little storage or processing power, it would be desirable to get some information to a larger computer, perhaps with a tape drive for long term storage. Currently, the Monitoring Station compresses all of the IP headers down to protocol, ring destination, ring source, length of packet and time of reception at a factor of 64:1 and sends these packets over the proNET ring to a VAX. This VAX can do much more complex analysis than can the PDP 11/20. Although sending compressed information to another computer seems like a reasonable idea, it might be better not to send packets over the network being monitored. An alternative would be to have the PDP 11/20 use either a serial line or a different network in order to transfer information. The best idea might be to have another mini-computer attached to the PDP 11/20. The PDP 11/20 would run the same software on a continuous basis. The other mini-computer could be used for real-time debugging and short-term network analysis running whatever software is useful at the time. At the same time, a tape drive would store all of the data generated by the PDP 11/20 on tape so that a complete set of records exist for later analysis. This later long-term analysis could be done on a VAX and since complete records exist, could be done any time any new information was required.

With the information collected, it is hoped that various parameters of ring operations and usage can be determined. Some of these include latency until transmission, number of defective packets, ring reliability, fairness and stability under high-load. Questions involving ring usage are protocols used, who is sending to whom, number of back-to-back packets, interpacket arrival time, distribution of packets size and distribution of usage throughout the day.

Currently, the Network Monitoring Station is running reliably but it still has some bugs in hardware and software that distort some of its statistics. Also a very elementary analysis program exists on the VAX for long term analysis. The proNET ring presently carries about 850 thousand packets and 140 million bytes on a busy day.

12. DIRSYS: AN ONLINE DIRECTORY ASSISTANCE SYSTEM

12.1. Overview

DIRSYS is an electronic telephone book. It was developed for users with widely varying computer skills. Therefore, the self-teaching aids for the novice were designed to not encumber the experienced user. It is based upon the familiar concepts of a paper phone book and a full-screen display editor such as EMACS. Entries from the directory database are displayed on the screen in a compact format, one line per entry, and DIRSYS indicates which is the current entry of interest by emphasizing the entry's line (capitalize all letters, filling in empty fields with periods, displaying in reverse video, etc.) The user may direct the system to emphasize another entry (i.e. move the system's pointer) by issuing commands, similar to EMACS' cursor motion commands, or by searching for a name. The search mechanism is incremental. That is, after each character typed by the user, DIRSYS updates its pointer and the terminal screen, if necessary, such that the pointer rests on the entry whose name string most closely matches what the user has typed so far. A help facility, in the form of a menu, is provided to guide the novice user and remind the experienced user what commands are available. The help facility operates in the same manner as the incremental interface, except the search mechanism has been removed. The default screen allows approximately a full screen's worth of entries to be displayed, each entry occupying one line of the terminal screen. All information concerning a particular entry cannot be seen using this compact format. The user may request DIRSYS to display much fewer entries on the screen and show each entry in detail. A command is available to switch between these two display formats. All commands retain their semantics regardless of the display format.

12.2. Current State of the Project

A prototype has been implemented on a DECSYSTEM-20 and is being moved to a VAX 11/750. The interface and database structure are to be evaluated and modified based on the evaluations. A mechanism for updating the database is being implemented.

References

1. Needham, R. and Schroeder, N. "Using Encryption for Authentication In Large Networks of Computers," Communications ACM, 21,12 (December 1978), 993-999.

2. Saltzer, J., Reed, D. and Clark, D. "Source Routing for Campus-Wide Internet Transport," Local Networks for Computer Communications, West, A. and Janson, P. (eds.), North-Holland Publishing Company, Amsterdam, 1980. 1-23.

PRECEDING PAGE BLANK-NOT FILMED

EDUCATIONAL COMPUTING

Academic Staff

H. Abelson, Group Leader

A. diSessa	S. Weir

Research Staff

A. Berger	D. Smith
G. Kiczales	S. Russell
L. Klotz	

Graduate Students

M. Eisenberg	J. Valente

Undergraduate Students

A. Cullen	R. Hyre
J. Kelly	E. Lay
J. Palevich	D. Scrimshaw
S. Strausman	E. Tenenbaum

Support Staff

E. Lawry	D. Tatar

1. INTRODUCTION

The goals of the Educational Computing Group are (1) to identify the mental models that novices employ in their interactions with formal and informal systems (2) to build flexible, easy-to-understand computational systems and (3) to combine our findings about the process of learning together with our computer systems to create educational environments for a wide range of subjects and students. Three complementary areas were the focus of our activities this year: the ongoing development of the Boxer system; a range of projects that allow us to work directly with teachers and students using existing computational tools; and continuing work with cognition and learning. We attempt to employ the useful ideas that students already have - about computers, about math, about language arts, and about physics - to give them an initial mastery over the computational situation while introducing new concepts gradually.

2. BOXER

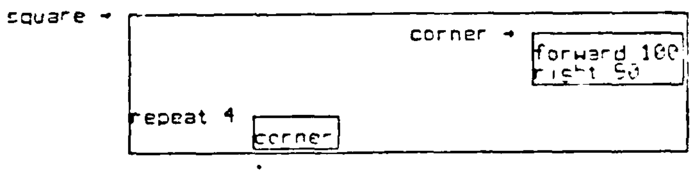
During the past year, we have continued the design and implementation of the Boxer computational environment, a system whose goal is to integrate a wide variety of applications - text manipulation, programming, data manipulation, communication, and graphics - within an easy-to-learn framework. Boxer is motivated by our conviction that most non-specialist users of computers are best served by providing a computational environment that is integrated and coherent, in which all of the basic capabilities can be assimilated to a single, uniform, easily understood computational scheme. In Boxer, we are trying to achieve system coherence by emphasizing (1) a single, spatial model to represent the hierarchical organization of programs and data; (2) the equivalence of objects and their screen representations; and (3) a uniform way of manipulating all system objects.

2.1. Design Principles of Boxer

One of our major concerns in formulating the design of Boxer is to provide a mechanism that will enable even beginning users to deal with the large-scale structure of the system. The approach we have adopted is to organize Boxer in terms of a pervasive spatial metaphor. Elements of the environment can be thought of as places, and their visible spatial relationships have structural meaning. All compound objects are versions of "boxes," which are two-dimensional arrays, possibly containing sub-boxes. The containment relation among boxes provides a uniform geometric metaphor that is used to model all hierarchical structures in the system, from variable scoping through program structure through file access.

Consequently, the entire system can be regarded as a two-dimensional geometric space through which the user moves. This is a crucial aspect of making the system accessible to beginning users, since it links the central organizing image of the system to geometric intuitions.

Figure 7-1 shows a simple example of spatial organization in a procedure called SQUARE, which draws a square on a graphics display screen by repeating four times a sequence of two moves called CORNER. Notice that the definition of the CORNER procedure is geometrically contained within the SQUARE procedure. This shows how boxes can be used to achieve the usual organization of block-structured languages. In Boxer, we extend this principle, using boxes as geometric carriers of many other hierarchical structures as well.



```
square →
                       corner → forward 100
                                 right 90
repeat 4 [corner]
```

Figure 7-1

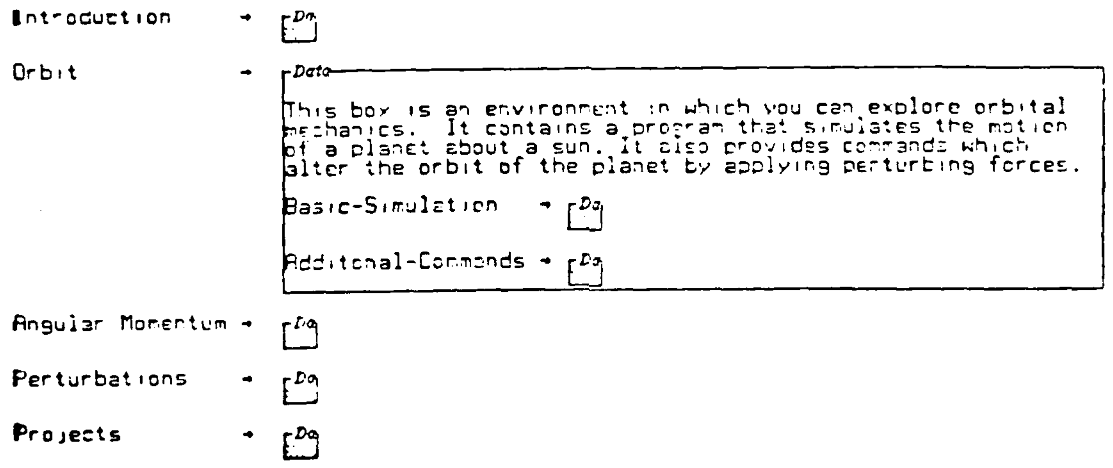
A second design principle in Boxer is the identification of objects with their screen representations. This is, in essence, a consistent commitment to the idea that the system *is* the way it appears on the screen. For example, any text that appears on the screen, whether typed by the system, typed by the user, previously executed or not, is available to be manipulated, edited, or executed. This uniformity enables Boxer to support styles of interactive use that are very different from those found in other computational environments. For example, one can execute statements one-by-one and then at some later time indicate that the typed statements should be incorporated into a program. Consequently, programming in Boxer is often not so much "writing" programs as it is piecing them together concretely from objects already in the system.

2.2. Focus of Our Work This Year

Our work over the past year has had two concerns. The first is to build a minimal, usable implementation on the Lisp Machine. The most difficult aspect of this is the display handling, in particular, the incremental redisplay algorithm for the Boxer screen editor. This is much more difficult than the redisplay for a conventional text editor because boxes are inherently two-dimensional structures whose sizes must dynamically adjust to fit their changing contents. A year ago, we built a prototype implementation that did not use incremental redisplay, and, even on the Lisp Machine, the resulting real-time interaction was so cumbersome that it was difficult to get a feel for what working in Boxer would be like. This problem was solved by Gregor Kiczales, who designed and implemented a sophisticated two-dimensional redisplay algorithm for Boxer. Although we still wish to explore various user-interface options (touch screens, pointing devices, and so on) the basic screen interaction is now adequate for our preliminary implementation.

Our second major concern this year was to begin to exploit Boxer's underlying uniformity to identify synergistic mixtures of programs, text, and data, in the recognition that Boxer's integrated framework provides makes it a novel and powerful computational medium. For instance many people have envisioned the possibility of using computers to create "interactive books," and there are a few systems that enable experts to implement these in limited contexts. In Boxer, this application falls out as a matter of course, since the system treats text and programs identically, making it straightforward to combine these in flexible ways.

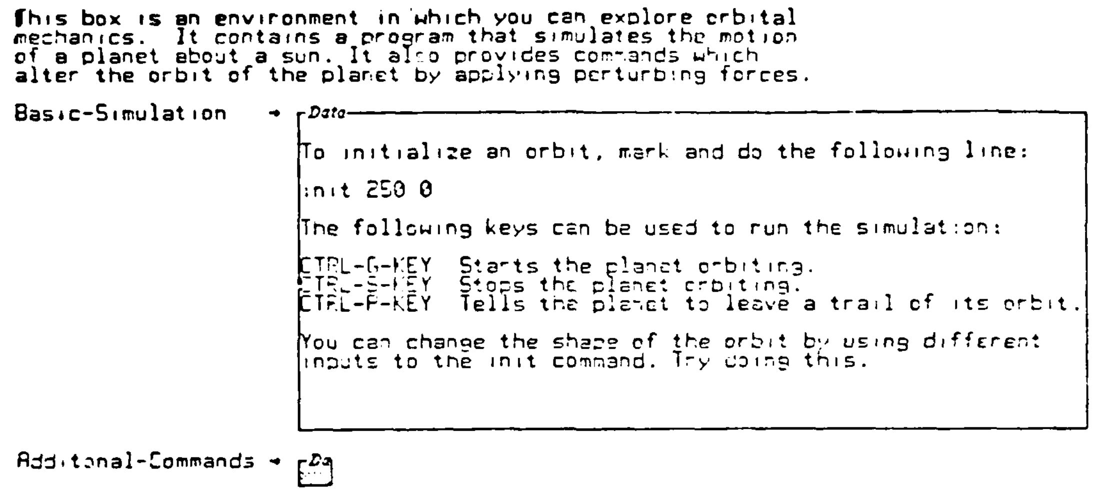
Figure 7-2 shows an example of such an interactive book in the area of orbital mechanics, written by Laura Bagnall. As shown, the box structure itself is used to organize the "book" into sections, so that a top-level view of the book, with the boxes shrunk, serves as a table of contents. In the figure, the ORBIT sub-box is expanded to show its interior detail.



```
Introduction          →  Da

Orbit                 →  Data
                         This box is an environment in which you can explore orbital
                         mechanics.  It contains a program that simulates the motion
                         of a planet about a sun. It also provides commands which
                         alter the orbit of the planet by applying perturbing forces.

                         Basic-Simulation    →  Da

                         Additonal-Commands  →  Da

Angular Momentum      →  Da

Perturbations         →  Da

Projects              →  Da
```

Figure 7-2

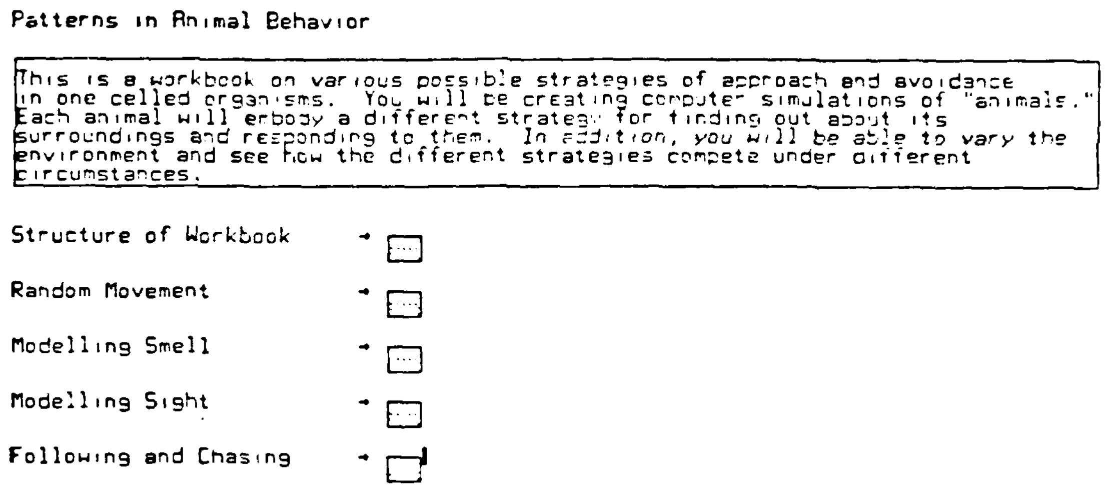
Figure 7-3 shows the complete ORBIT sub-box, which itself contains a box called BASIC-SIMULATION. This box contains not only text, but also a simulation program that enables users to experiment with various gravitational orbits.



```
This box is an environment in which you can explore orbital
mechanics.  It contains a program that simulates the motion
of a planet about a sun. It also provides commands which
alter the orbit of the planet by applying perturbing forces.

Basic-Simulation    → Data
                      To initialize an orbit, mark and do the following line:

                      init 250 0

                      The following keys can be used to run the simulation:

                      CTRL-G-KEY  Starts the planet orbiting.
                      CTRL-S-KEY  Stops the planet orbiting.
                      CTRL-P-KEY  Tells the planet to leave a trail of its orbit.

                      You can change the shape of the orbit by using different
                      inputs to the init command. Try doing this.

Additonal-Commands → Da
```

Figure 7-3

Figure 7-4 shows the top level of another interactive book. This one, written by Deborah Tatar, is in the area of mathematical biology. It contains programs that model mechanisms of directed movement used by simple animals. Figure 7-5 shows a section of the book that deals with random motion. In addition to the explanatory text, this box contains a procedure called WALK, which simulates random motion and also, at the bottom of the screen, a sequence of commands that can be executed to run the simulation. In addition to executing the command as shown, the user is also free to edit the command and thus experiment with the effects of changing parameters. Another area we have been exploring this year is the use of Boxer for simple information retrieval. Figure 7-6 illustrates the use of pattern-directed data manipulation, in a system implemented by Eric Tenenbaum. In the figure, we see a box called STUDENTS, which contains typical student record information for an undergraduate course. The nested box structure serves to organize the information hierarchically - each sub-box of STUDENTS contains the information for a single student which is divided into a NAME part, an ADMIN part, and a GRADES part. In addition to the records, STUDENTS contains a local procedure called NEW-STUDENT-TEMPLATE, which, when executed, produces a new sub-box with the appropriate fields for new student information. Figure 7-7 shows a box, HIGH-SCORES, which is meant to be a filter on student records. This filter, written in a simple pattern matching notation, will select out the student



Patterns in Animal Behavior

This is a workbook on various possible strategies of approach and avoidance in one celled organisms. You will be creating computer simulations of "animals." Each animal will embody a different strategy for finding out about its surroundings and responding to them. In addition, you will be able to vary the environment and see how the different strategies compete under different circumstances.

Structure of Workbook →

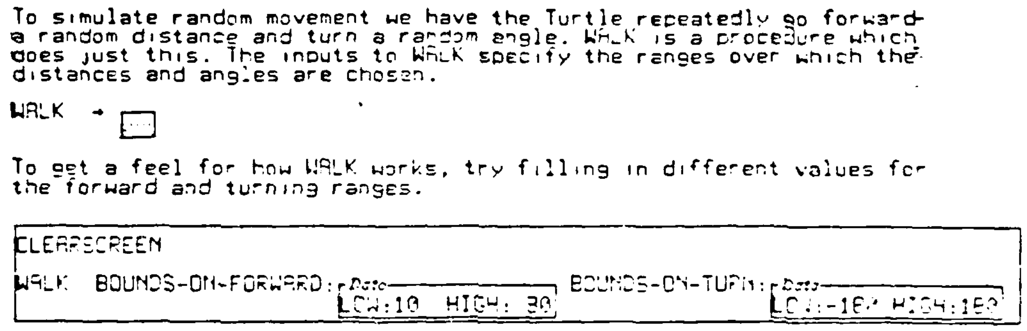
Random Movement →

Modelling Smell →

Modelling Sight →

Following and Chasing →

Figure 7-4



To simulate random movement we have the Turtle repeatedly go forward a random distance and turn a random angle. WALK is a procedure which does just this. The inputs to WALK specify the ranges over which the distances and angles are chosen.

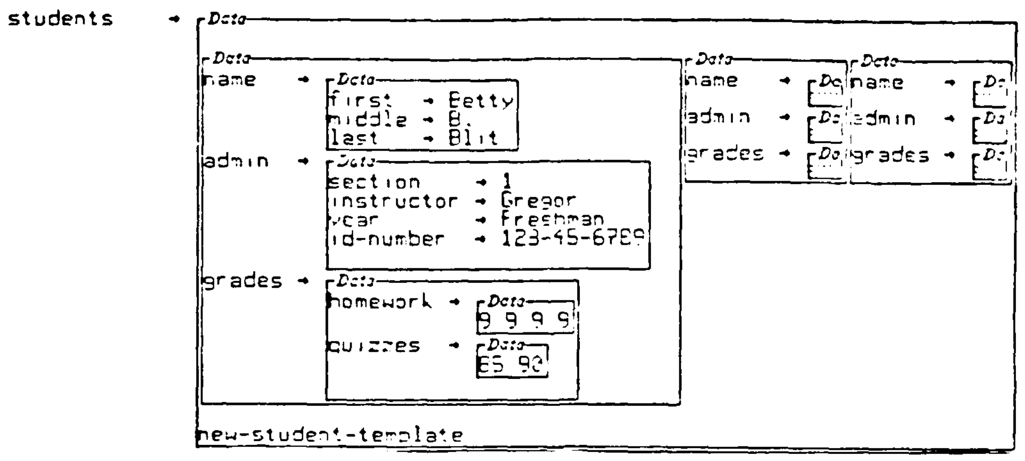
WALK →

To get a feel for how WALK works, try filling in different values for the forward and turning ranges.

```
CLEARSCREEN
WALK  BOUNDS-ON-FORWARD: Data                 BOUNDS-ON-TURN: Data
                         LOW:10  HIGH: 30                     LOW:-180 HIGH:180
```

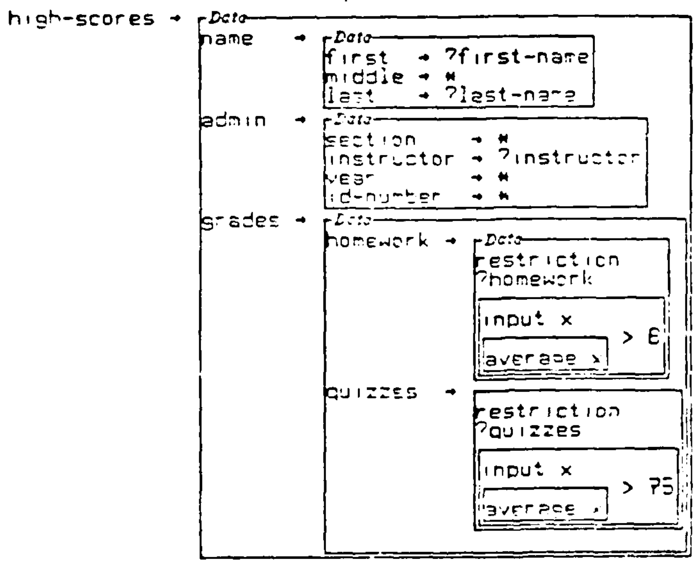
Figure 7-5

records in which the average of the homework grades is greater than 8 and the average of the quiz grades is greater than 75. Figure 7-8 shows how this pattern is used together with a command COLLECT to run through the STUDENTS data base, applying the filter HIGH-SCORES. For each record that passes through the filter, the program generates a copy of the form EXCELLENT-STUDENT-FORM, with the designated variables filled in using corresponding information from the record. As with the interactive books, we hope to show that a small set of features, such as COLLECT, when embedded in the Boxer environment, will make substantial data-manipulation capabilities available even to beginning users of the system.



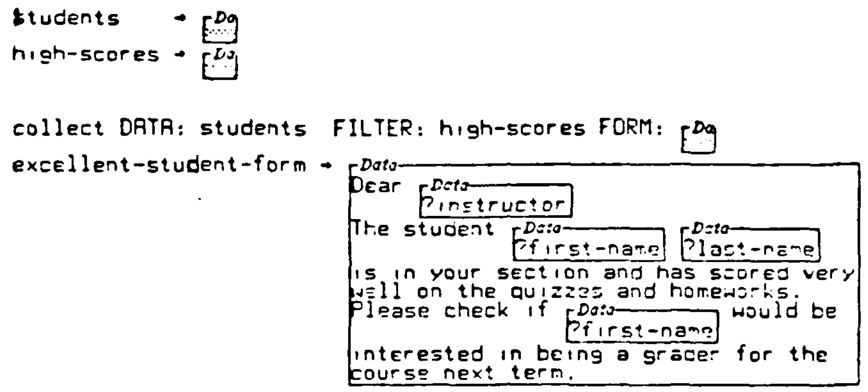
```
students  → Data
  Data
  name   → Data
             first  → Betty
             middle → B.
             last   → Blit
  admin  → Data
             section    → 1
             instructor → Gregor
             year       → Freshman
             id-number  → 123-45-6789
  grades → Data
             homework → Data
                          9 9 9 9
             quizzes  → Data
                          85 90
  Data                     Data
  name   → Da             name   → Da
  admin  → Da             admin  → Da
  grades → Da             grades → Da
new-student-template
```

Figure 7-6



```
high-scores → Data
  name   → Data
             first  → ?first-name
             middle → *
             last   → ?last-name
  admin  → Data
             section    → *
             instructor → ?instructor
             year       → *
             id-number  → *
  grades → Data
             homework → Data
                          restriction
                          ?homework
                          input x
                          average x  > 8
             quizzes  →   restriction
                          ?quizzes
                          input x
                          average x  > 75
```

Figure 7-7



```
students    → Da
high-scores → Da

collect DATA: students  FILTER: high-scores FORM: Da
excellent-student-form → Data
                         Dear Data
                              ?instructor
                         The student Data         Data
                                     ?first-name  ?last-name
                         is in your section and has scored very
                         well on the quizzes and homeworks.
                         Please check if Data         would be
                                         ?first-name
                         interested in being a grader for the
                         course next term.
```

Figure 7-8

3. WORK WITH STUDENTS AND TEACHERS

During the past year, Sylvia Weir has continued two strands of activity begun while part of the MIT Division for Study and Research in Education: one strand concerns the special opportunities offered by the presence of computers in the classroom; the other concerns the problems associated with a large-scale introduction of computers into the educational system.

3.1. Logo and Special Populations

The Logo system is being used with special populations in three ways. As well as its primary use as an educational tool, Logo serves a diagnostic function, facilitating the observation of the strengths and weaknesses of individual children to determine their needs; and as a research instrument, to expand our understanding of children's thought processes by examining the way in which they perform Logo-based activities. For example, using standard Logo and modified versions of Logo, computer-based ways of assessing the learning needs of children with a variety of physical handicaps have been developed, and appropriate curricular materials to match those needs have been created (with Russell, Valente and Berger at the Cotting School for Handicapped Children). Work with autistic and emotionally disturbed children is aimed at finding ways of optimizing Logo for a cognitively low functioning population (with Scrimshaw at the League School). For both these populations, the development of an updated Periman button-box input device that plugs into the Apple's game I/O connector, allows Logo commands to be input by the depression of a single key, making the process conceptually simpler, and reducing the number of gestures and accuracy of physical control required.

Introducing Logo into the curriculum of 5 - 12 year-old dyslexic children (with Kelly at the Carroll School) has highlighted the role of instructional choices in fostering learning problems. Our schools place an overly strong emphasis on language-based activity in the classroom, and this has put those individuals whose gifts happen to lie in the spatial domain at a disadvantage. Prior to their demonstrated proficiency in Logo, neither dyslexic individuals nor their teachers appreciate the possibility that their spatial skills may be relevant to formal school or other testing situations. There is a growing appreciation of the role of spatial problem-solving in math understanding. In many disciplines such as, for example, in some branches of engineering or in organic chemistry, one is required to visualize, to form an image relating underlying mechanisms to some observed or desired behavior. The computer introduces, for the first time, the possibility of allowing such individuals to show their intellectual strengths at an earlier age than has been the case until now, since connecting a computer graphics-screen to a learning environment can support formal problem-solving in the spatial domain.

Spatial skills involved in Turtle Geometry include the estimation of length and degree of turn; the exercise of a visual imagination, enabling the projection of meaningful images on to sparse half-drawings; an ability to see and exploit symmetry; and a ready response to visual feedback. For example, we see children who can readily estimate half the length of a line in turtle units, but cannot divide by two; who can estimate the perimeter of a regular hexagon in turtle units just by looking at it, without being supplied by numbers, but by "eyeing it" - their explanation. Yet they are unable to generate the answer to the equivalent question given as a standard mathematical sentence. One can divide a line in half, in a Logo setting, by visually deciding where the half point is, and separately deciding how many turtle units correspond to the length of that half-line. Nowhere has one actually applied the arithmetic operation: divide a number by two. Building bridges between this ability to perform spatial reasoning and more conventional mathematical methods becomes the next educational task.

3.2. Teacher Training

There is a widespread concern about the crisis in teacher-training in the educational uses of computers. It is our experience and that of other Logo educators that the innovative nature of our approach exacerbates the problem.

Logo is intended to be more than a programming language, and the creation of varieties of computational environments to support learning in other curricular areas requires special training. There is a particularly urgent need for the exhibition of good ways to use Logo, and for on-going support over long periods, to enable a school system remote from immediate contact with a community of Logo users to install and manage a Logo-based computer center, and to integrate Logo activities into the school curriculum. In response to this perceived need for models, we have been producing a series of videotape modules which makes a start in this direction (with Smith, Russell, Berger and Valente). Videotapes will be accompanied by supplementary written materials, and, in some cases, by demonstration programs on floppy disks.

A further step involves a plan to introduce an electronic communication network into an educational community to structure the required support system.

In all these activities, we look forward to the guidance and support of Seymour Papert, who has joined our Group.

Publications

1. Abelson, H. Logo for the Apple II, Peterborough, NH, Byte Books, 1982.

2. Abelson, H. Apple Logo, Peterborough, NH, Byte Books, 1982 (alternate edition of *Logo for the Apple II*).

3. Abelson, H. "A Beginner's Guide to Logo," Byte Magazine, (August 1982).

4. diSessa, A. "Learning Physics from a Dynaturtle," (with B.Y. White) BYTE Magazine, (August 1982).

5. diSessa, A. "Phenomenology and the Evolution of Intuitions," in Mental Models, D. Gentner, and A. Stevens, (eds.), New Jersey, Lawrence Erlbaum Press, 1983.

6. diSessa, A. "A Principled Design for an Integrated Computational Environment," MIT/LCS/TM-223, MIT Laboratory for Computer Science, Cambridge, MA, July 1982.

7. diSessa, A. "Intuition as Knowledge," Progress Report for the Spencer Foundation, Chicago, IL, December 1982.

8. Russell, S.J. "Delta Drawing: Another View," Classroom Computer News, (May/June 1983).

9. Russell, S.J. "Logo: An Approach to Educating Disabled Children," with S. Weir and J. Valente, Byte Magazine, (September 1982).

10. Scrimshaw, D. "The League School Report," MIT, Internal Memo, Cambridge, MA.

11. Scrimshaw, J.D. "Logo and Autistic Children," MIT, Internal Memo, Cambridge, MA.

12. Weir, S. "Logo: A Learning Environment for the Severely Handicapped," Journal of Special Education, (1982).

13. Weir, S. "The Computer as a Creative Educational Tool," American Annals of the Deaf, 127(5), (1982), 690-2.

14. Weir, S. "Logo: An Approach to Educating Disabled Children," Byte Magazine 7(9), (1982), 342-60.

Theses Completed

1. Lay, E. "An Interactive Optics Workbook," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1983.

2. Palevich, J. "DANDY--An Expandable Real Time Adventure," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1982.

3. Strausman, S. "Implementing Learning Tools for Primary Mathematics Using Sprite Logo," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1983.

4. Tenenbaum, E. "Pattern Matching and Data Manipulation in the Boxer System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1983.

Talks

1. Abelson, H. "Turtle Geometry," 10th Annual Conference on Mathematics and Statistics, Miami University, Oxford, OH, October 1982.

2. Abelson, H. "Turtle Geometry," Conference on the Third Hemisphere, Florida State University, Tallahassee, FL, March 1983.

3. Abelson, H. "Turtle Geometry," Fourth Annual Conference of the Educational Computing Organization of Ontario, Ontario Institute for Studies in Education, Toronto, Canada, April 1983.

4. Berger, A. "Using Logo with the Physically Handicapped," Education Collaborative, Brookline, MA, December 1982.

5. Berger, A. "Microcomputers and The Language Arts," with S.J. Russell, Conference on Media and Computers, International Reading Association Regional Meeting, Lexington, MA, October 1982.

6. Berger, A. "Microcomputers and Writing," The Writing Project, Institute for Professional Development, West Roxbury, MA, January 1983.

7. diSessa, A. A series of five talks on computers in education, World Center for Informatics and Human Resources, Paris, France, July 1982.

8. diSessa, A. "Turtle Geometry - Inside and Outside Logo," Department of Mathematics, University of Montana, Missoula, MO, October 1982.

9. diSessa, A. "Mental Models of Programming Languages," Department of Computer Science, University of Stuttgart, Stuttgart, Germany, February 25, 1983.

10. diSessa, A. "Intuitive Physics," Faculty of Psychology and Education, University of Geneva, Geneva, Switzerland, March 2, 1983.

11. diSessa, A. "Designing an Understandable Computational Environment," Faculty of Psychology and Education, University of Geneva, Geneva, Switzerland, March 8, 1983.

12. diSessa, A. "Science Education in a Computational Environment," Department of Mathematics/Computer Science, Southeastern Massachusetts University, North Dartmouth, MA, March 23, 1983. Also a second lecture, "Computers in the Future of Education."

13. diSessa, A. "The Computer as an Epistemological Catalyst," Annual Meeting of the American Educational Research Association, Montreal, Canada, April 15, 1983.

14. diSessa, A. "Intuitive Physics," Department of Psychology, Vanderbuilt University, Nashville, TN, April 19, 1983.

15. diSessa, A. "Educational Assumptions of Logo," Swedish National Commission on Computers in Education, Cambridge, MA, May 31, 1983.

16. diSessa, A. "Future Computer Languages for Education," Swedish National Commission on Computers in Education, Cambridge, MA, June 4, 1983.

17. Kiczales, G. "Boxer - An Integrated Personal Computing Environment,"
    Lawrence Livermore High Speed Computing Conference,
    March 1983;
    Educational Computing Conference of Ontario, March 1983.

18. Russell, S.J. "Logo and Mathematical Ideas in the Elementary School," American Association for the Advancement of Science Annual Meeting, Detroit, MI, May 1983.

19. Russell, S.J. "Logo as a Tool for Assessment and Curriculum Planning," Bureau of Cooperative Educational Services Conference on

Microcomputers and Special Populations, Westchester County, NY, May 1983.

20. Russell, S.J. "Logo and Special Learning Needs: the Teacher's Role," Lesley College Annual Conference on Computers in Education, Cambridge, MA, May 1983.

21. Russell, S.J. "If There Were Worlds Enough and Time: Microcomputers and Special Education," New England Regional Conference on Computers in Education, Kennebunkport, ME, April 1983.

22. Russell, S.J. "Logo with the Physically Handicapped," Educational Computing Group of Ontario, Fourth Annual Conference, Toronto, Canada, April 1983.

23. Russell, S.J. "Technology for Everyone," Keynote address, Council for Exceptional Children, Rockport, ME, November 1982.

24. Russell, S.J. "Microcomputers and the Language Arts," with A. Berger. Conference on Media and Computers, International Reading Association Regional Meeting, Lexington, MA, October 1982.

25. Russell, S.J. "Using Young Children's Spontaneous Play," with F. Fisher. Conference on Families and the Nuclear Crisis, Wheelock College Center for Parenting Studies, Boston, MA, October 1982.

26. Weir, S. "Logo for the Physically Handicapped," Center for Cognitive Science, University of Western Ontario, London, Ontario, Canada, April 8, 1983.

27. Weir, S. "Information Prosthetics for the Handicapped," Thames Valley Children's Centre, London, Ontario, Canada, April 9, 1983.

28. Weir, S. "A Computer Environment for the Future," Communications Consortium, Inc., MIT, Cambridge, MA, May 4, 1983.

29. Weir, S. "Logo and Children with Special Needs," Developmental Evaluation Clinic Seminar, Children's Hospital Medical Center, Boston, MA, May 6, 1983.

30. Weir, S. "Games for Remediation," Video Games and Human Development Conference, Harvard Graduate School of Education, Cambridge, MA, May 22 - 24, 1983.

FUNCTIONAL LANGUAGES AND ARCHITECTURES

Academic Staff

Arvind, Group Leader

Research Staff

R. Thomas

Graduate Students

D. Culler	V. Kathail
P. Fuqua	P. Lim
S. Heller	G. Papadopoulos
R. Iannucci	K. Pingali
	R. Soley

Undergraduate Students

T. Chambers	T. Huffman
R. Chu	C. Ozveren
M. Dovek	R. Sathyananda
S. Ghani	K. Suh
M. Hoang	R. Wei

Support Staff

K. Warren

1. INTRODUCTION

The primary direction of the Functional Languages and Architectures Group has not changed from last year. We continue to study new computer structures to exploit the parallelism in functional programs. Our approach in studying parallelism is based on a highly dynamic interpreter for dataflow graphs called the U-interpreter [4]. We believe the success of a general-purpose multiprocessor computer depends on its effective programmability and its efficient utilization of resources, and accordingly, we are concerned with hardware issues and with associated system problems such as high level language support, communication requirements, and efficient distribution of workload over the machine.

We are well along in developing the high level dataflow language Id and have an operational Id-to-graph compiler for our prototype dataflow machine. The process of specifying the functionality of each component of the prototype machine in enough detail to facilitate hardware implementation is complete and is being tested in a variety of 'softer' environments: simulation and emulation.

We feel the development of this novel architecture will require several iterations. Hence, my Group is pursuing a variety of interrelated projects, all aimed at developing our understanding of the problems and moving us closer to a final implementation. First, we have written a simulator for the Tagged Token Dataflow Machine to run on an IBM 4341. Second, we are constructing an emulated dataflow machine by connecting 64 Symbolics 3600 (LISP) machines. This will include the development of language and system software to make the emulated machine usable by applications programmers. Both the simulated and the emulated machine will use the graphs generated by the Id compiler. The system software work is just starting. In parallel we are investigating the decomposition of programs and also developing techniques for efficient code generation from functional languages with streams. We think these techniques for code generation are also applicable to sequential and other non-dataflow parallel machines.

2. ARCHITECTURE

2.1. Principles

Attention has been focused in the recent past on constructing multiprocessor systems [11][12][13][15][16][8] -- attention derived from a desire for more performance, greater fault tolerance, the ability to exploit the rather curious economics of a single-chip computer, or whatever. What has *not* been done sufficiently is a re-evaluation of the assumptions that led us to where we are now. This is painfully clear when one observes that von Neumann style uniprocessors still form the building block for the majority of multiprocessor projects or proposals.

Many variations on the von Neumann theme have been explored (e.g., pipelining, multiple functional units, vector instructions), but little has been done with the sequential control required for instruction execution. There are good reasons to believe that this re-evaluation is overdue.

Arvind and Iannucci have put forward three fundamental architectural issues on which to base this re-evaluation: scalability, tolerance of memory latency, and the ability to share data without constraining parallelism [6]. *Scalability* requires simply that the system can be made incrementally more powerful by adding more hardware resources, and without reprogramming. *Latency* is the length of time between issuing a memory request and getting a response. While the architects of high performance machines have always paid attention to memory latency issues, their solutions, notably caches, are not directly applicable to parallel machines. Most of the early attempts at designing parallel machines, and some of the new architectures, have not dealt with this problem adequately. Consequently, achieving high performance through parallel processing has remained an elusive goal. The Tagged Token Data Flow Architecture was designed with these issues in mind. It is tolerant of memory latency by virtue of the basic programming model. I-structure storage provides the ability to process memory requests out of order and to share data without constraining parallelism.

2.2. Developments

The details of the architecture have evolved substantially in this past year. In particular the functional breakdown into modules, the program structure, and the addressing facilities have been completely specified [3].

Our machine is a hardware implementation of the U-interpreter for a graphical data flow base language [4]. Programs written in any functional language which can be compiled into this base language may be executed on our machine. Such a compiler for the high-level data flow language Id is already in use. The U-interpreter uncovers parallelism in programs during execution by uniquely labeling independent activities as they are generated. Each instance of execution of an operator is called an *activity*, and is given a unique *activity name*. Activities that have all their input values available can execute provided a processor is available.

An abstract U-interpreter machine has five essential subsystems

1) a *waiting-matching* section,

2) an *instruction-fetch* section which is connected to a program memory,

3) an *arithmetic logic* unit,

4) an *output* section, and

5) a *data structure* storage.

These subsystems are connected as shown in Figure 6-1. The first four subsystems form a ring in which many tokens may circulate in a pipelined fashion. Data structure storage provides an alternative path to the other four sections. Assume that every token carries, in addition to an activity name, data and port number, the number of partners needed to enable the destination instruction. Now according to the rules of the U-interpreter the operands for an activity must carry identical tags. The *waiting-matching* section matches input tokens with their partners. If a match is found and the corresponding activity is *enabled* then matched tokens are forwarded to the *instruction fetch* section; otherwise the incoming token is stored in the *waiting-matching* section. All code blocks reside in the program memory. The *instruction fetch* section retrieves the instruction specified by the code block name and statement number part of the activity name from the *program memory*. Each instruction contains an *opcode* and the addresses of the instructions to which the results are to be sent. The opcode from the instruction and the two operands are passed to the ALU which produces the result data value and passes it to the *output* section. The output section computes the new activity name for the result data value, in accordance with the U-interpreter rules, using the input activity name and the destination instruction address stored in the program memory. An output token is produced for each of the destination instructions specified in the current instruction. If the ALU executes a data structure operation (e.g., *append* or *select*) , a token for the data *structure storage* is generated and routed appropriately. The tokens for the *data structure* storage are different from the tokens that enter the *waiting-matching* section because they carry an opcode (e.g., read, write) and do not require matching of activity names or instruction fetching.

A multiprocessor U-interpreter machine can be formed by replicating the abstract machine of Figure 6-1 and connecting these machines by a packet communication network. Programs are mapped on such a multiprocessor by assigning activities and data structures to individual abstract machines (henceforth called Processing Elements or simply PE's). Since in any implementation the time to send a token from one PE to another PE would be significant, the advantage of distributing activities uniformly in space and time over PE's can be overshadowed by non-local program and data references. There is no global memory in the design we have chosen for the multiprocessor machine, therefore the instruction corresponding to an enabled activity must be available in the local memory. This is achieved by loading a block of code on a group of PE's prior to the execution of the code. A code block for which this type of pre-loading makes sense is associated with a *logical domain* which is either a *loop domain* or a *procedure domain*. Since a logical domain is the set of tokens generated in one invocation of a loop or procedure, the tokens belonging to it

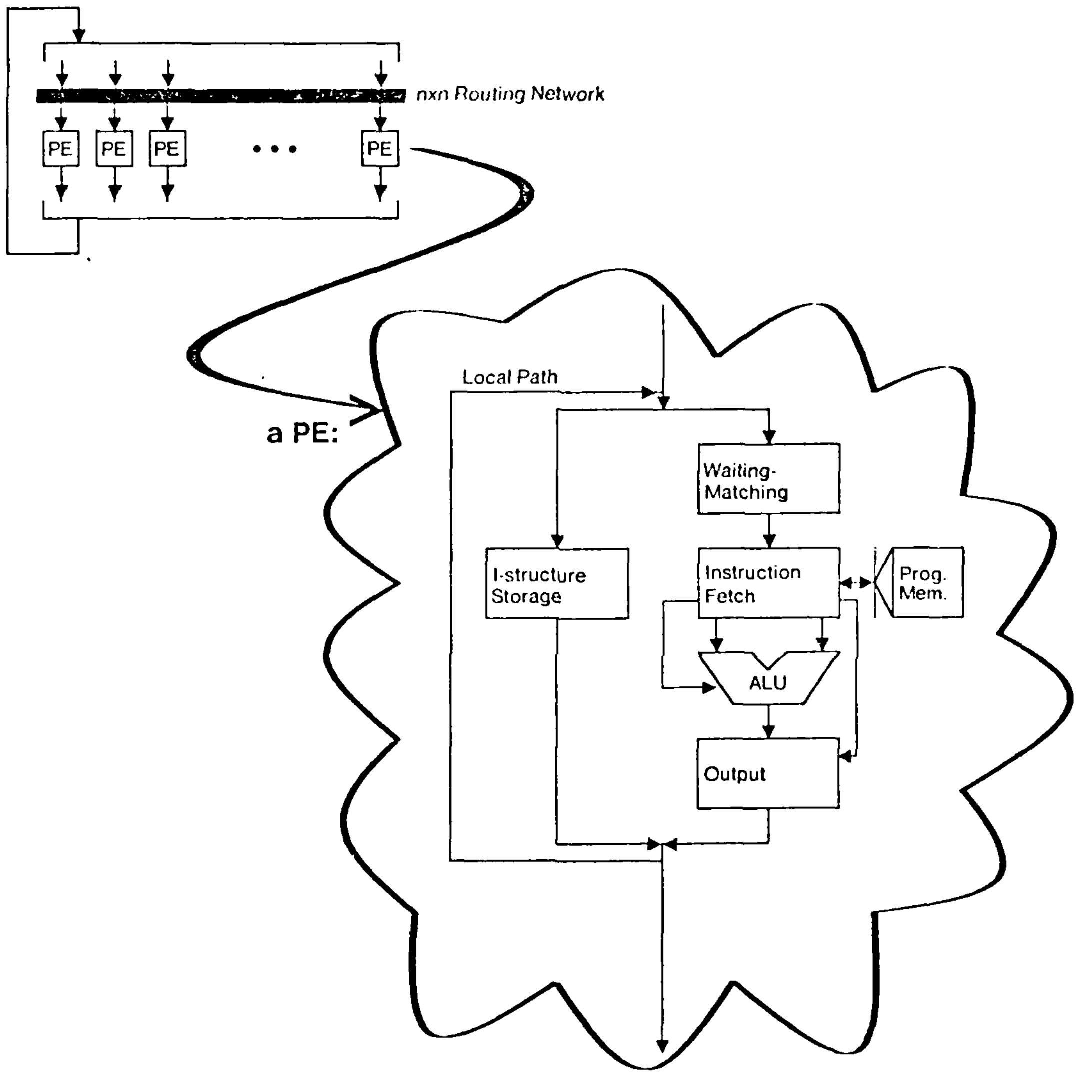


Figure 6-1: A Block Diagram of the Abstract Machine

should be mapped onto a group of PE's in which intragroup communication distances are as short as possible. We call such a group of PE's a *physical domain*. For simplicity we assume that a physical domain is characterized by a base PE number, PE_{base} and a size υ, and PE's with numbers PE_{base} to $PE_{base} + \upsilon - 1$ belong to it. After the machine has been (logically) configured into several *physical domains*, the mapping of programs can be done in two steps:

1) Assignment of a procedure or loop at the time of its invocation to a physical domain. This includes allocation of data structure and program storage for the invoked code block.

2) Assignment of activities of a logical domain within a physical domain.

Since the total number of activities is, in general, orders of magnitude larger than the total number of logical domains for a program the second mapping should be performed with minimal time overhead. However, selection of a physical domain for a newly created logical domain can be done by a semi-centralized scheduler which can reside on any group of PE's. Operators that initiate loops and procedures (L and A operators), when executed, send a code block name and some scheduling parameters to the scheduler so that it can select an appropriate physical domain. Schedulers are Id *manager* programs [4][2] which will permanently reside in the machine. The policies to be employed by these managers are issues under current investigation.

Activities within a physical domain can be assigned to PE's by a simple hashing of statement numbers, initiation numbers or both [7]. For example, the destination PE number for the tokens destined for an activity with statement number s and iteration number i can be calculated as PE_{base} + (s-1)mod υ. This will imply that statement numbers 1, $1+\upsilon$, $1+2\upsilon$ ... etc. should be loaded on PE_{base} , statements 2, $2+\upsilon$, $2+2\upsilon$... etc. should be loaded on PE_{base} + 1 and so on. Thus the code will get uniformly distributed over the physical domain. However, with only one copy of the code, all initiations of a statement will execute on one PE. This may severely restrict the exploitation of parallelism in a program. Mappings based on initiation counts may show very different behavior. Suppose the destination PE number is calculated as PE_{base} + (i-1)modυ or as PE_{base} + (i-1/k)modυ where k is the number of consecutive passes of a loop that should be kept within the same PE, then each PE in the physical domain will need a complete copy of the code, and up to υ initiations of a statement may execute simultaneously. It is also possible to combine the two schemes and choose a subdomain (i.e., a group of PE's with a complete copy of the code) on the basis of the initiation number while the PE within the subdomain is chosen on the basis of the statement number.

In [1] we have discussed a general strategy for decomposing programs. Following that, we would like to minimize the distance from the PE that creates a data structure element to the PE that uses that element. Often a logical domain creates a data structure that the next related logical domain needs.[5] In such a condition, mapping of both logical domains on the same physical domain can avoid a lot of unnecessary data transfers. It should be noted that the efficiency of a data structure mapping is closely related to the assignment of instructions such as *select* and *append*, that manipulate the data structure.

The architecture of the PE for the Tagged Token Dataflow machine is closely related to the architecture of the abstract U-interpreter machine and is shown in Figure 6-2. The PE (see Figure 6-2) has eight asynchronously functioning subsystems which are connected by finite size buffers and communicate with each other using a send-acknowledge protocol. The special service processor marked *PE control* has access to the memory of all subsystems via a common bus, and is used for diagnostic and special memory functions. The data structure storage is designed to support I-structures efficiently. Although the activity names generated by the U-interpreter may become arbitrarily large, in the Tagged Token Machine all activity names are represented by finite size *tags*. Therefore tags have to be reused as computation progresses. The machine only knows about tags and does not explicitly represent the complete activity names. It achieves this by manipulating tags in a way which is isomorphic to the way the U-interpreter manipulates activity names. Tag manipulation is related to the token and instruction formats (the instruction set of the machine is discussed in [5]), the address translation scheme, and the mechanism to support procedure calls. This is specified in [3].

The tokens entering a PE can be divided into the following three groups:

1) **Tokens corresponding to values in data flow graphs:** These are associated with the conceptual notion of tokens (i.e., tokens that move about on the arcs of the dataflow graph).

2) **System-generated tokens:** These are closely tied to the specific implementation and have no direct interpretation in terms of dataflow graphs. Tokens in this class do not enable instructions; rather, they carry with them the necessary operation codes.

3) **Tokens for PE control and diagnostic purposes:** These types of tokens are used to convey information in and out of the PE control subsystem to the host processor.

Each type of token has a different format and follows a different path inside the PE.

[5]Logical domains u and u' are related if u = w.c.s.i and u' = w.c.s.i + 1.

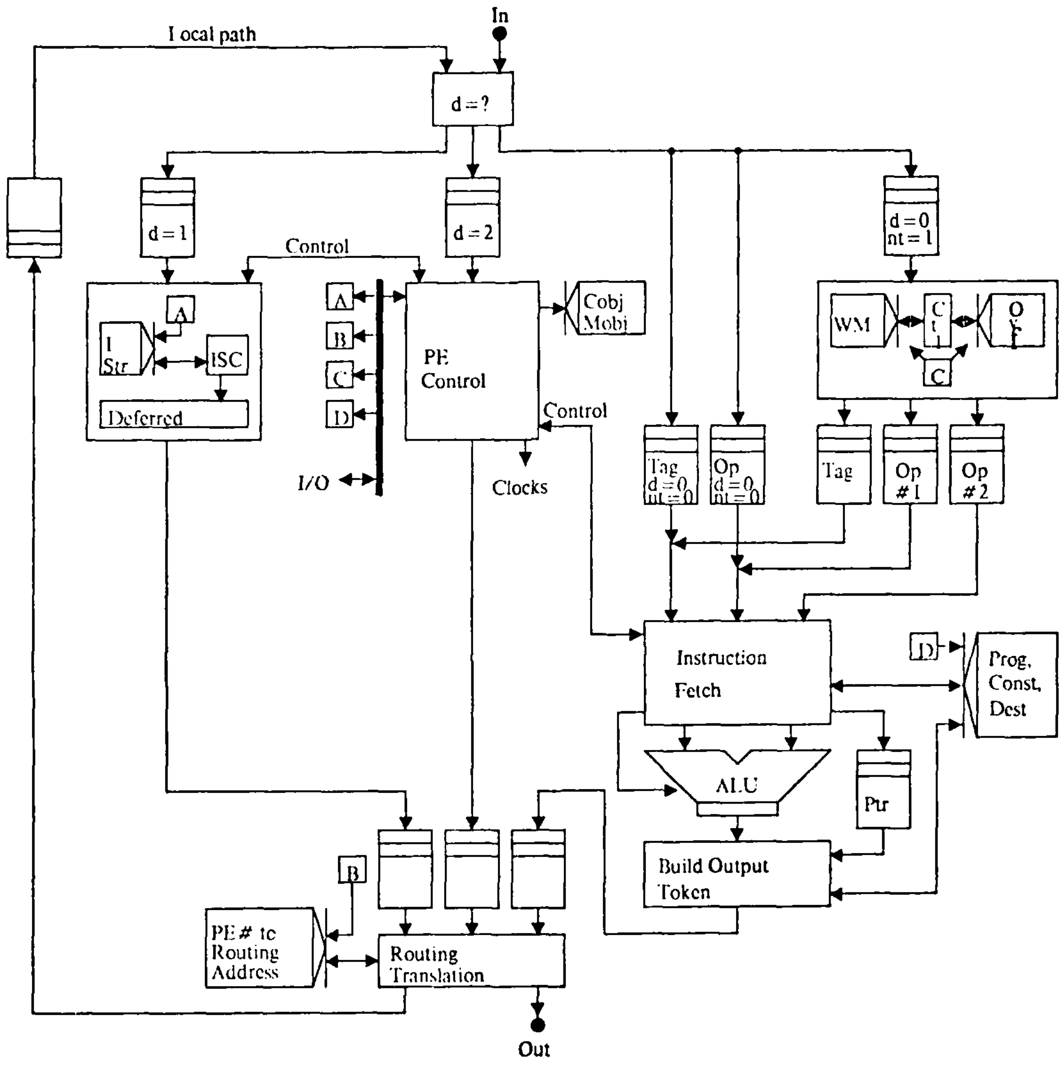


Figure 6-2: A Block Diagram of the PE

But one can view the generation of each type of token as a consequence of executing an instruction in the ALU.

2.3. Simulation

A detailed simulation of the Tagged Token Dataflow Architecture has been developed by Keshav Pingali, David Culler, and Moris Dovek, with some early contributions by Arvind and Robert Thomas. This involves modeling the functional and temporal behavior of every component of the machine, as well as the interaction of components through finite buffers. The basic program is operational, but various interfaces are still in progress. We hope to start simulation experiments to study the dynamic behavior of dataflow programs in the summer 1983. IBM Research is cooperating in these experiments, running the same program on one or more larger machines at Yorktown. The program is currently 200 pages of Pascal code and is likely to double in size by the time experiments will begin. The goal is to execute at least 20 million dataflow instructions per run which may take as much as 24 CPU hours in stand alone mode on the IBM4341.

The simulated analogue of the architecture is described in Figure 6-3. The ovals correspond to hardware components, each of which is implemented in the simulation as a PASCAL procedure. The boxes correspond to finite buffers connecting the various components. The simulation program comprises these component procedures, procedures for modeling the behavior of the buffers, a system manager, and an overseer that dispatches activities and interfaces the various components. It is an event driven simulation, with a variety of optimizations to reduce the overhead of the overseer. This facility allows us to experiment with all the low level details of the architecture and test various approaches to program decomposition. By virtue of its detail, however, we expect this facility to be too slow to run application programs of substantial size.

2.4. Emulation Experiments

It is very important, at this stage, to test the applicability of the dataflow approach on large applications. For this reason we are developing a fairly high speed emulated machine, built out of 64 Symbolics 3600 symbol processors. The hardware to interconnect the machines is currently under development and should be available for testing in the Fall of 1983. The emulator will be written originally in LISP and then gradually migrated to the microcode of the 3600 processor. The LISP version of the emulator should be ready by December 1983. Assuming the funding and procurement of equipment goes smoothly, we should begin testing the Tagged Token Dataflow Machine on an ensemble of four to eight Lisp Machines early next year. The goal is to efficiently execute real applications on the 64 machine configuration by the end of 1985.

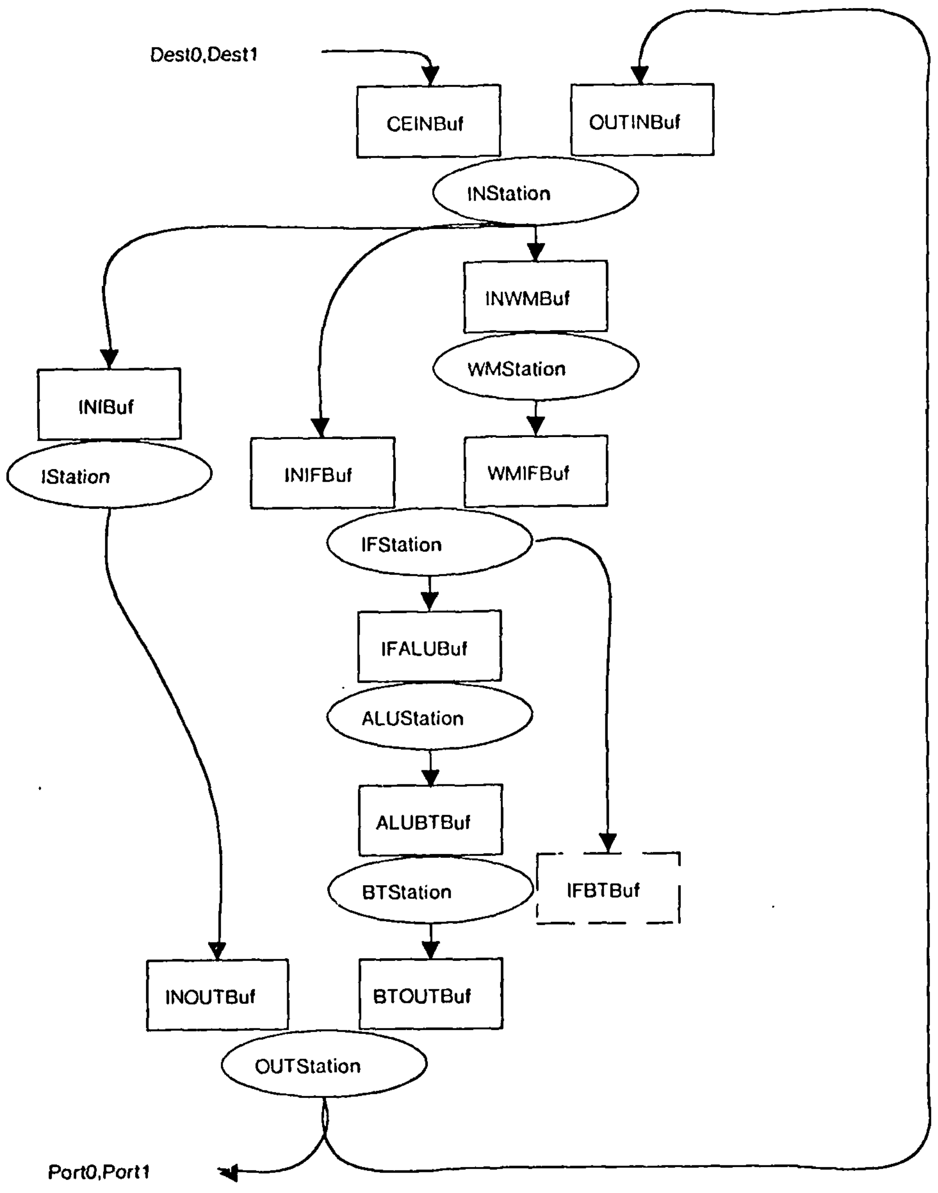


Figure 6-3: Analogue of the Architecture

This emulated machine is a single application of a general emulation facility under development at LCS. Section 4 will discuss this facility and the emulated machine in detail.

2.5. Hardware

Our final goal is, of course, to translate this architecture to hardware. Concomitant with this goal, we must keep up with the developing technology. In the past, various parts of the Tagged Token Data Flow Machine were designed as custom VLSI chips. This year two hardware projects were carried out.

In Steve Heller's thesis [9] an I-structure memory controller was developed using the IDL (Interactive Design Language) system. This was a very large hardware project, raising issues that will challenge the hardware design languages of the future. The ISMC design required 1500 lines of IDL code and involves a PLA with:

- 887 Product terms
- 151 Inputs (excluding feedbacks)
- 316 Output (excluding feedbacks)
- 91 Feedbacks

This design was transliterated into PASCAL for the simulation program by Robert Chu, Richard Wei, and Tim Chambers.

Robert Iannucci investigated the design and simulation of a VLSI circuit for Manchester encoding and decoding of a serial data stream. The design employs a phaselock loop for recovering the clock from the encoded data, and is capable of running at a rate of 100 Megabits per second when implemented with the 1.2μm. n-well cMOS process.

3. LANGUAGES

3.1. Demand Driven Evaluation

In his engineer's thesis titled "Efficient Demand-driven Evaluation", Keshav Pingali described a method for performing demand-driven evaluation on dataflow machines. The basic idea is to transform dataflow programs such that a data-driven evaluation of the transformed program performs exactly the same computation as a demand-driven evaluation of the original program. The transformation essentially involved the explicit modeling of demands, and permitted the compiler to introduce demand propagation operators into programs. The technique suggests a simple denotational semantics in the style of Kahn for demand-driven evaluation. He also described a source-to-source transformation of dataflow programs such that the overhead of propagating demands in the transformed program is less than the corresponding overhead in the original program. This transformation is also relevant for sequential implementations of applicative languages.

3.2. Program Decomposition

The problem of decomposing programs in a multiprocessing system is longstanding. Dataflow principles simplify the problem because the basic sequencing mechanism guarantees correct operation regardless of the mapping onto the physical processors. Nonetheless, the problem is difficult and this research is still in the early stages.

David Culler investigated the problem of program decomposition in a dataflow system, focusing on nested loops. It was shown that a collection of loops nested K deep with simple index functions can be carried out by successively executing K-1 degree hyperplane of operations simultaneously. This is a generalization of transforming the program into a sequential loop containing K-1 parallel loops on a control sequenced system.

A simple model of communication behavior was put forward in order to study a variety of specific problems and decompositions. This substantiated our intuition that both execution behavior and communication behavior must guide the decomposition and that the communication costs increase substantially with the degree of parallelism exploited. This study will be extended to deal with the situation where two large portions of the computation cooperate (e.g., a producer-consumer relationship) or operate independently (effectively competing for resources).

3.3. Compiler Development

Development of the compiler to translate Id to the machine code for the Tagged Token Dataflow Machine continued during the last year. Vinod Kathail and Ki Suh have completed the machine code generator which forms the last phase of the code generation process. It translates the dataflow graphs to the actual instructions for the machine as described in [5], and implements the procedure call mechanism described in [3].

Vinod Kathail incorporated an algorithm to detect deadlocked cycles as well as dead code in Id programs into the compiler. The cycle detection algorithm uses a modified version of the cycle-sum test of Wadge [14]. The search for the cycles is performed in the direction opposite to the direction of the data flow arcs, thereby allowing us to detect dead code while searching for cycles.

Robert Stanzel completed a preliminary version of the I-structure detection program. His program can detect I-structures for the case when they are generated by a *for* loop and the selectors at which appends are performed are of the form a*i + b where a and b are absolute constants and i is the loop index. We have also developed a reference count scheme for I-structures. Since the generation and consumption of an I-structure is usually done by a loop, the scheme tries to reduce the instructions used to increment or decrement reference counts by assuming that each code-block (i.e., a procedure or a loop) consumes *one* reference to an I-structure. Reference count is incremented when a reference to an I-structure is either passed to another code-block or stored in another structure. Reference counts are decremented at the completion of the code-block. Certain optimization rules allow us to further reduce the number of instructions by grouping them together. An augmented version of the I-structure detection algorithm that also determines the size of an I-structure and the reference count scheme will be incorporated in the compiler during this summer.

In addition, we are investigating ML like type structure for Id.

4. MULTIPROCESSOR EMULATION FACILITY

Emulation is a powerful pre-construction evaluation technique which allows the system architect to abstract away from some of the low-level details of an implementation so that attention may be focused on the higher level behavior of the system being designed. We define *emulation* as the process of reconfiguring a *base* computer system via programming, so that it behaves according to a *target* architectural specification. The programming may take the form of micro-coding, higher-level programming, or some mixture. To make the base machine behave according to the architecture of the target machine means that the base machine will accept the target machine-level programs and data structures, and will produce results identical to those of the target architecture, although not necessarily with the same level of performance. Unlike a *simulation* program, an emulator usually does not carry timing information explicitly.

4.1. Rationale

The *emulation facility* is a system consisting of 64 powerful computers and 64 8 x 8 network switches that interconnect these machines. These computers and switches can be programmed either through a high-level language (so as to reduce programming effort) or at the micro-code level (for increased performance) to emulate a variety of multiprocessor systems. The emulation of such multiprocessor machines prior to their construction is deemed essential for the following reasons.

1) Analytical/theoretical or single-processor simulation techniques alone

are not adequate for solving a variety of problems that must be addressed prior to implementation of a multiprocessor system, such as determining the size of buffer memories in each processor.

2) The alternative of implementing a proposed multiprocessor architecture directly with custom silicon chips is costly, time consuming and good for testing only one architecture at a time.

3) The emulation facility *fosters* a multiprocessor design orientation -- in effect, the designers of multiprocessor systems become accustomed to "thinking parallel" precisely because their experimental base is the parallel structure of the emulation facility.

4) The availability of such a powerful emulation facility for use by members of the DARPA community will stimulate the design of numerous architectures by top-level designers while simultaneously scrutinizing the potential success of these machines at a small fraction of the implementation costs.

The proposed emulation facility is made up of two integrated parts --- the *microprogrammable processor* and the *packet communication switch module*. The former is available commercially; the latter is not. The facility will consist of 64 total processors, eight of which will be fully-configured as software development stations (with display, keyboard, paging disk, and local network interface). The other 56 processors will be stripped down versions (with memory but without display/keyboard or disk). From the software point of view, all 64 machines will be programmed as if they were identical for emulation purposes. Each processor, fully configured or not, will have an integrated module. These switches will be interconnected under user control to a variety of inter-processor communication network configurations.

The software for the Multiprocessor Emulation Facility (MEF) will be written almost entirely in the LISP language to allow us the maximum flexibility in adapting to future changes. Since the 3600 processor was designed specifically with the LISP language in mind (unlike almost all other commercially available processors), we expect little speed impairment because of this choice.

A small although vital part of the emulation facility software will be hand coded in 3600 microcode. Included in this will be the code necessary to access the high-bandwidth inter-processor network of MEF via the 3600's main processor bus called the *L-Bus*. Besides making use of this network for transmitting dataflow tokens from one machine to another, we intend to implement a facility whereby LISP level software can view the entire primary memory of MEF as one unified address space.

This would be done by mapping local memory addresses (i.e., interpreting the higher order eight bits as a processor number) onto the real memory of the local or a foreign network

4.2. Network Switch

The design of the network is built around the concept of *packet switching* which was developed under DARPA funding. In such a network, a processor formats information to be transmitted into data packets by the addition of some routing and control information (analogous to putting a letter into an addressed envelope) and hands it off to the network. The sending processor is then free to perform other work while the network goes about the task of delivering the packet to the proper destination. Such networks vary in their interconnection topology, but have the property that a packet may go through several links. Rather than reserving a complete path from source to destination for the entire duration of the packet's transit time called *circuit switching*, such networks allow packets to be *stored* at intermediate points until a path is available to the next intermediate point. The network then *forwards* the packet and frees the storage space used for the next packet.

Robert Iannucci and Robert Thomas have designed a switch that will form the basic element of this network. The proposed switch implements extensive failure detection through analog and digital domain checks, and relies on the associated microprogrammable processor for error recovery operations. This approach reduces the complexity of the switch, while simultaneously raising the level of fault tolerance. The switch has also been systematically designed to *minimize* the chance for failure by relying only on robust techniques.

The overall structure of the switch is shown in Figure 6-4 (see also [10] for more details). It is modular in character, and is organized around eight major control and data buses. Connected to the buses are eight input FIFOs, eight output buffers, and a Sequencer/Scheduler. Seven of the input FIFOs and seven of the output buffers connect to serial -- parallel conversion logic. The eighth input/output pair interfaces to the data bus of the attached microprogrammable processor. Internally, the switch is totally synchronous. Resynchronization of incoming data with the local clock is done by the serial-to-parallel conversion logic.

Robert Thomas has been involved in the hardware design and implementation of a 32 M bit/second serial communication link for the MEF. This link has recently been powered up and should be debugged by the end of June. The link is an extremely important part of the switch design since it will be replicated many times to implement the high-speed interprocessor connections.

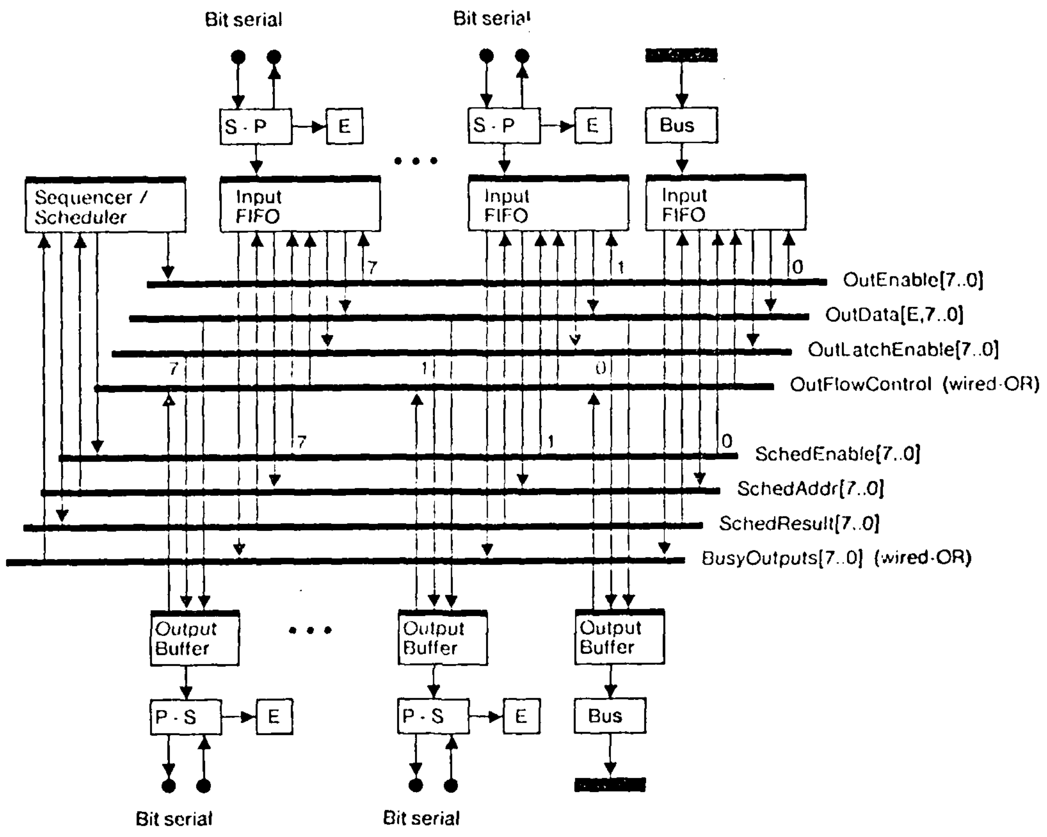


Figure 6-4: Basic Structure of the Packet Switch

In support of the hardware design of the switch, he wrote a general-purpose micro assembler which can be used to generate micro code for a class of finite state machines (useful for hardware controllers, e.g., the serial link has two such controllers). Also, he supervised two bachelor theses. One compared several hardware designs for generating check bytes that are appended to each packet in the communication network for error detection. In the other thesis, an algorithm was designed and implemented in a simulator to establish the topology of a distributed set of interconnected processors where each processor's only source of information about the other processors is from the single network input port connected to it (each processor also knows how many output ports it has and uses them to generate traffic on the network to derive the topology). This algorithm should save a good deal of manual work which would otherwise be needed to update network routing tables whenever the topology of the network changes either by design or hardware failure.

4.3. Emulation System

Once the emulator is established we envision several ways for making it useful to other researchers. One approach would be to duplicate the machine in whole or in part. This effort would only involve fabrication of the necessary switch modules with no additional design time.

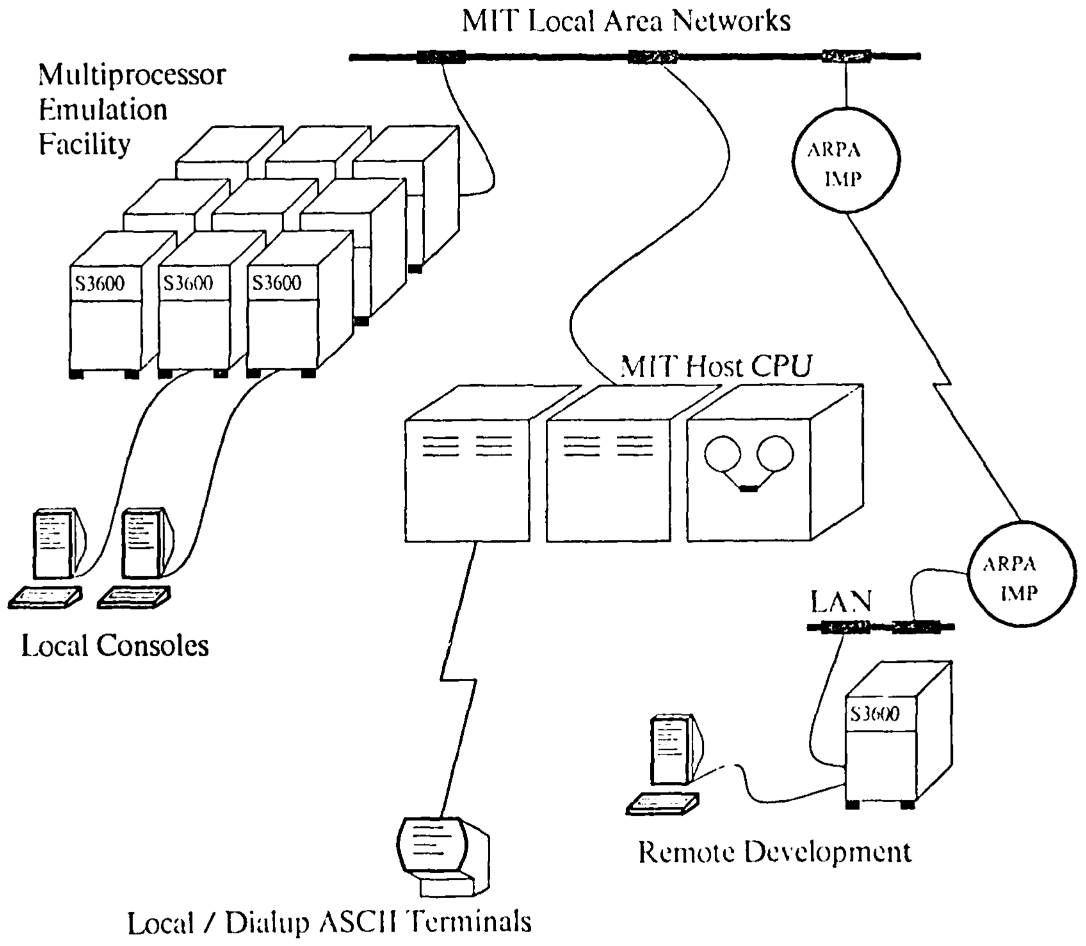


Figure 6-5: Use of the Multiprocessor Emulation Facility

Alternatively, the target interpreter could be developed remotely on another

Symbolics 3600, Symbolics LM-2, LMI Lambda, or MIT CADR and then transmitted via ARPANET to MIT for processing on our facility. (See Figure 6-5.) The emulation results would be then communicated in reverse to the remote user.

Due to the flexibility of the Symbolics 3600 hardware, it is possible to keep complete memory core images of several target interpreters at the emulation facility site. This flexibility would allow usage of the emulation facility by researchers throughout the United States who have some access to the ARPANET.

Richard Soley has suggested a scenario for remote usage of the emulation facility: A researcher designs, generates, and debugs code for the facility at the remote site, using another Symbolics 3600, Symbolics LM-2, LMI Lambda, or MIT CADR Lisp Machine. Since the dialects of Lisp provided by these four machines are approximately compatible, this presents no translation hardship at a later date. When the researcher's program is completed, it is transmitted to the emulation facility maintainers (via the ARPANET) in the form of a compiled LISP program or a group of programs.

At the MIT emulation site, a "simple" 3600 machine core image is developed and stored. This simple core image will contains only the bare minimum to implement the LISP interpreter and compiler, together with the lower-level primitives of the emulation system. Upon reception of a researcher's system (*i.e.*, target interpreter), a new core image is built from this base and the user's program. This core image is then loaded into each of the processing elements of the system, and emulation is activated.

This solution to the problem of outside usage of the system was chosen for its natural simplicity and flexibility. It allows the researcher tᐢ develop programs in a leisurely way, on a foreign host. Remote users need only know the barest minimum about the facility itself, primarily that it runs standard Lisp Machine LISP with some added primitives but withᐢout any program development tools (such as an editor, disassembler, etc.).

It is our intention to provide users of the facility with the necessary primitives with which to construct a target interpreter. Each user will start with the view of the emulator as 64 separate but interconnected processors. It will then be up to the user to piece together the supplied routines for remote memory reference or message-based communication to form the target system.

We believe that our proposed plan will lead to a simple, easy-to-use interface to the emulation facility by multiple-processor machine architecture researchers across the country.

4.4. Experiments

The first experiment will be an emulation of the dataflow architecture. Because of the complexity of the target architecture, Richard Soley has devised a three stage strategy. Stage one will be the implementation of the complete architecture on the emulator, but it will be written entirely in LISP. By doing so. we retain a significant degree of flexibility at a time when it is most crucial, i.e., the early testing and evaluation of the tagged-token machine.

One implication of this approach is that data would be stored in *simulated* memory, i.e., LISP arrays. Operations passed between the major blocks of the dataflow processing element will be represented by *flavor instances*, or at least *structures*, to give us the maximum coding simplicity and flexibility. An eye to future migration and to tractability (for performance testing) is necessary at this stage.

Since LISP-simulated storage will, in the long run, be too costly in terms of access time penalty for our purposes, the second stage would be to use the innate 3600 store. In addition, since most of the processors in the completed machine will not have virtual memory, this is the first stage in which we can run real computations. PE memory will be represented directly by *wired* pages, which the LISP code deposits into and reads from with the standard 3600 micro-coded primitives. We should be able to set aside wired, non-garbage-collected, untouched-by-LISP, *permanent* pages to hold I-structure memory, program memory, and the waiting-matching store. Most or all of the code at this stage, however, would still be in LISP.

The last stage of code migration would be the translation of some of the LISP code into micro-code. Before we come to this point, many test runs will be executed to give us a good feel for *hot-spots*, i.e., Lisp code fragments which are executed often and which can be speeded up by micro-coding.

Due to the complexity of the tagged-token architecture, we are estimating mean code paths on the order of 500 to 5000 micro-instructions per emulated machine cycle (including the overhead of those functions implemented in Lisp rather than in micro-code). This implies that the facility will interpret dataflow graphs at a rate of approximately 64,000 to 640,000 instructions per second.

It should be clear that the proposed emulation facility will be substantially easier to construct than a VLSI version of the intended target systems (especially the Tagged-Token Dataflow Machine). "Cost" of construction, aside from the necessary financial support, involves designing, building, testing, and replicating the packet switch module along with the necessary micro-code routines for using the switch and for error recovery.

The PE Controller (which controls the operation of a single dataflow processing element) and the storage emulation code was written by Richard Soley. Paul Fuqua contributed the I-structure controller. Soley wrote the skeleton for the ALU which is being completed by Poh Lim. This fall we hope to have the microcoded network token access functions operational so that we may begin to test our facility in a multiprocessor mode.

A great help to writing code for MEF in the absence of Symbolics 3600 processors has been the existence of the numerous MIT CADR Lisp Machines which run virtually the same source code as the 3600. (Our first two 3600's arrived in late May 1983.) This fortunate state together with our choice of LISP as our implementation language has added greatly to the ease with which code can be written, tested, and installed.

5. RELATED TOPICS

5.1. Fault Tolerance

Dataflow models have been developed to analyze and synthesize redundant computer systems which are able to tolerate hardware faults. The intended application is for systems which demand *provably correct*, very high levels of reliability, such as the real-time control of aircraft, spacecraft, and critical processes. In the work of Gregory Papadopoulos, current results in reliable systems research have been unified under a rigorous dataflow formulation. Algorithms have been developed to transform nondeterministic programs, written assuming perfect hardware, into a redundant model of a system that can tolerate a specified number of hardware faults and still maintain the full functionality of the original program. This permits an unambiguous understanding of important implementation assumptions: the number of independent hardware modules required, the nature and topology of the interconnection between modules, the synchronization required, and the algorithms which ensure consistent inputs to all redundant operators.

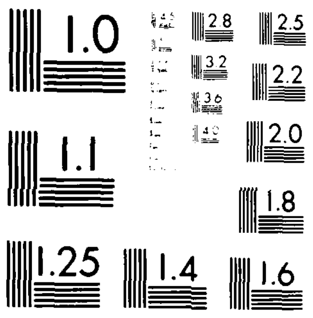
Central to the approach is the model for the propagation of hardware faults throughout the system. Once operators have been grouped into *fault sets* corresponding to failure-independent hardware modules, fault propagation from one module to another is strictly through the exchange of information between modules and over program edges. A faulty operator is permitted to produce completely arbitrary outputs, and even give *different* outputs to each operator that is connected to it.

Even though this fault model is very general, analysis of the system is greatly simplified since synchronization and control mechanisms are wholly specified by the flow of data over program edges. Moreover, the inherent absence of any

AD-A145 134   MIT LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 20 - JULY 1982 - JUN 1983(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE M L DERTOUZOS

UNCLASSIFIED   JUL 84 N00014-83-K-0125   F/G 9/2   NL

2/3


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

requirements for centralized and/or synchronous control is an extremely appealing foundation for the engineering of highly reliable systems.

5.2. New Equipment

A new computer room was established to house the equipment for the emulation facility, as well as some of the existing machines in the building, and a s/370 compatible IBM machine. This is a 4341 model group 2 on three year loan from IBM for the purpose of running the simulation program. It is equipped with a large memory system, 16 megabytes of primary storage and 2.4 gigabytes of secondary storage, to suit this application. It will run the VM/SP operating system. We plan to connect it to the MIT net and indirectly to the ARPANET (TCP/IP).

5.3. Computer Aided Engineering

As further support for hardware design, Robert Thomas evaluated several commercially available "computer aided engineering" design stations as well as the schematic capture systems currently available at MIT. The Idea 1000 system by Mentor Graphics Inc. emerged as a clear choice and we expect two such design stations to be operational by July 1983. This system offers schematic capture, logic simulation, timing verification, formatable net listing; future options include gate array design and circuit simulation (i.e., Spice) integrated with the other tools.

References

1. Arvind "Decomposing a Program for Multiple Processor System," Proceedings of the 1980 International Conference on Parallel Processing, August 1980, 7-14.

2. Arvind and Brock, J.D. "Streams and Managers," MIT-LCS-TR-217, MIT Laboratory for Computer Science, Cambridge, MA, June 1982.

3. Arvind and Members of the Functional Languages and Architectures Group, "The Architecture of the Tagged-Token Dataflow Machine," Computation Structures Group Memo, MIT Laboratory for Computer Science, Cambridge, MA, March 1983.

4. Arvind, Gostelow, K.P. and Plouffe, W. "An Asynchronous Programming Language and Computing Machine," Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, CA, December 1978.

5. Arvind and Iannucci, R.A. "Instruction Set Definition for a Tagged-Token Data Flow Machine," Computation Structures Group Memo 212, MIT Laboratory for Computer Science, Cambridge, MA, December 1981. Revised February 1983.

6. Arvind and Iannucci, R.A. "A Critique of Multiprocessing von Neumann Style," Proceedings of the 10th International Symposium on Computer Architecture, June 1983.

7. Gostelow, K.P. and Thomas, R.E. "Performance of a Simulated Dataflow Computer," IEEE Transactions on Computers C-29, 10, 905-919, (October 1980).

8. Gottlieb, A., Grishman, R., Cruskal, C.P., McAuliffe, K.P., Rudolph, L. and Snir, M. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Transactions on Computers C-32, 2, 175-189, (February 1983).

9. Heller, S.K. "An I-Structure Memory Controller," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

10. Iannucci, R.A. "Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility," Computation Structures

Group Memo 220, MIT Laboratory for Computer Science, Cambridge, MA, October 1982.

11. Sullivan, H. and Bashkow, T.R. "A Large Scale, Homogeneous, Parallel Machine," Proceedings of the The Fourth Annual Symposium on Computer Architecture, March 1977.

12. Swan, R.J., Fuller, S.H. and Siewiorek, D.P. "Cm* - A Modular Multiprocessor," Proceedings of the National Computer Conference, 1977.

13. Swan, R.J., Bechtolsheim, A., Lai, K-W. and Ousterhout, J. "The Implementation of the Cm* Multi-microprocessor," Proceedings of the National Computer Conference, 1977.

14. Wadge, W.W. "An Extensional Treatment of Dataflow Deadlock," In G. Kahn (ed.), Lecture Notes in Computer Science, 70: Semantics of Concurrent Computation, Springer-Verlag, (1979), 285-299.

15. Widdoes, L. "The S-1 Project: Developing High-Performance Digital Computers," Proceedings of COMPCON Spring 1980, February 1980, 282-291.

16. Wulf, W.A., Levin, R. and Harbison, S.P. Hydra/C.mmp: An Experimental Computer System, McGraw-Hill, 1981.

Publications

1. Arvind, and Iannucci, R.A. "Critique of Multiprocessing Von Neumann Style," Proceedings of the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June 1983.

2. Bauman, N.B. and Iannucci, R.A. "A Methodology for Debugging Data Flow Programs," Computation Structures Group Memo 219, MIT Laboratory for Computer Science, Cambridge, MA, October 1983.

3. Iannucci, R.A. "Single-chip Microtask Scheduler for a Dataflow Machine," FLA Design Note #1, MIT Laboratory for Computer Science, Cambridge, MA, August 20, 1982.

4. Iannucci, R.A. "Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility," Computation Structures Group Memo 220, MIT Laboratory for Computer Science, Cambridge, MA, October 1982.

5. Pingali, K.K. and Arvind "Efficient Demand-driven Evaluation(1)," submitted to ACM TOPLAS.

6. Pingali, K.K. and Arvind "Efficient Demand-driven Evaluation(2)," submitted to ACM TOPLAS.

7. Thomas, R.E. "Micro Assembler User's Guide for Finite State Machines," FLA Design Memo, MIT Laboratory for Computer Science, Cambridge, MA, February 27, 1983.

Theses Completed

1. Pingali, K.K. "Efficient Demand-driven Evaluation," S.M. and E.E. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

2. Seltzer, W.A. "A Communication System for a Multiprocessor Dataflow Computer Architecture," S.M. and E.E. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1982.

3. Heller, S.K. "An I-structure Memory Controller (ISMC)," S.M. and E.E. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

4. Papadopoulos, G.M. "Data Flow Models for Fault-Tolerant Computation," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

5. Hoang, M.N. "Self-Initialization Protocol for Multiprocessor Networks," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

6. Huffman, T.E. "Error Detection Circuit for a Packet Communication Switch," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

7. Varkonyi, R.B. "Low Altitude Aircraft Measurement System Real Time Record Program Development," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

Theses in Progress

1. Seth W.M. "Optimized Execution of the APL Structured Functions," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

2. Fuqua, P. "Emulating the I-Structure Memory for a Tagged-Token Dataflow Processing Element," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

3. Soley, R. "The Emulation of a Tagged-Token Dataflow Processing Element," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

Talks

1. Arvind "A Dataflow Architecture with Tagged Tokens," Digital Equipment Corporation, Hudson, MA, October 7, 1982.

2. Arvind "Dataflow and Signal Processing," USC-ONR Workshop on Modern Digital Signal Processing, University of Southern California, Los Angeles, CA, November 2, 1982.

3. Arvind "A Dataflow Architecture with Tagged Tokens," California Institute of Technology, Pasedena, CA, November 3, 1982.

4. Arvind "A Dataflow Architecture with Tagged Tokens," UCLA, Los Angeles, CA, November 4, 1982.

5. Arvind "A Dataflow Architecture with Tagged Tokens," University of California, Irvine, CA, November 5, 1982.

6. Arvind "Limitations of Multiprocessors based on Conventional Processors," Los Alamos - Livermore Workshop on Parallel Architectures for Scientific Computing, Boulder, CO, January 25, 1983.

7. Arvind "A Facility to Experiment with New Architectures," ICL, Windsor, UK, February 11, 1983.

8. Arvind "Id: A Dataflow Language," Workshop on SAL, Digital Equipment Corporation, Hudson, MA, February 17, 1983.

9. Arvind "Dataflow Analysis and Id," IBM, Yorktown, NY, February 22, 1983.

10. Arvind "Dataflow and High Performance Computers," Fermi Laboratory, Industrial Associates Program, Fermi Laboratory, Batavia, IL, May 19, 1983.

11. Arvind "A Dataflow Architecture with Tagged Tokens," Dataflow Symposium, University of Utrecht, Utrecht, Netherlands, June 2, 1983.

12. Arvind "A Dataflow Architecture with Tagged Tokens," Brown Bovari Co, Baden, Switzerland, June 6, 1983.

13. Arvind "A Dataflow Architecture with Tagged Tokens," Royal Institute of Technology, Stockholm, Sweden, June 7, 1983.

14. Arvind "A Dataflow Architecture with Tagged Tokens," Ericsson, Stockholm, June 8, 1983.

15. Arvind "What is Demand-Driven Evaluation," Royal Institute of Technology, Stockholm, June 9, 1983.

16. Arvind "What is Demand-Driven Evaluation," Chalmers Institute of Technology, Gothenborg, June 12, 1983.

17. Arvind "A Critique of Multiprocessing von Neumann Style," The 10th International Symposium on Computer Architecture, Stockholm, June 17, 1983.

18. Iannucci, R.A. "Data Flow Computer Architecture: An Exercise in Top-Down Design," July 22, 1982.

INFORMATION MECHANICS

Academic Staff

E. Fredkin, Group Leader

Research Staff

T. Toffoli	G. Vichniac

Graduate Students

N. Margolus

Undergraduate Students

S-W. Chen	T. Ketudat

Support Staff

R. Hegg	A. Schmitt

1. CONSERVATIVE LOGIC AND REVERSIBLE COMPUTING

One of the goals of conservative logic is to explore ways of realizing virtually nondissipative computing. To this purpose it is necessary to achieve a very close match between the logical structure of the desired computation and the structure of the physical computer that should carry it out.

In previous years we had constructed combinational and classical-mechanical models of nondissipative computation, and we had started working on quantum-mechanical models.

1.1. Quantum-Mechanical Computation

In the past year, we have constructively shown the existence of systems that obey the principles of quantum mechanics and that are capable of carrying out reversible, nondissipative computation. A student of ours, Norman Margolus, has been working in close contact with Richard Feynman, who's been exploring an implementation of this idea based on concrete physical effects.

1.2. Information-Mechanical Reduction of Physical Concepts

The past years have seen the definite acceptance of the view that physical entropy is but a particular case of "information-theoretical entropy," a concept which is meaningful for systems of a much more general nature, such as symbolic dynamical systems, data structures, etc. We suspect that other quantities until now presumed to be peculiar of physical systems actually admit of a much more general interpretation; among these, energy and temperature. We have started studying this problem, using ideas from conservative logic—especially the billiard-ball model of computation—and from the dynamics of reversible dynamical systems—in particular, reversible cellular automata.

1.3. CAM—A High Performance Cellular Automata Machine

We have made much use of CAM, the cellular automata machine developed within our Information Mechanics Group. The demos that we have given have raised considerable interest in this kind of simulation techniques. Pending the resolution of administrative problems, we are in the process of setting up a CAM Consortium of 15—20 members who are interested in owning a copy of the machine and share software, application programs, and results.

Gerard Vichniac has conducted extensive investigations of models of percolation, nucleation, and annealing. He has shown that many statistical-mechanical concepts pertaining to these models (e.g., order parameters, nonergodicity, nonseparability, order-disorder transitions) can in fact be defined purely in terms of primitives of computational and informational nature, such as counting, labeling, and comparing. We therefore hope that these concepts can find fruitful applications in computer science, much like entropy did.

Norman Margolus has devised a clever cellular-automaton rule that implements conservative logic in an extremely compact and efficient way. With this rule, we can simulate on CAM computations involving thousands of signals and gates.

1.4. Scientific Exchanges

The whole group participated with a number of contributions to the Cellular Automaton Workshop held at Los Alamos in March 1983. Tommaso Toffoli was chosen as one of the three proceedings editors.

Demos of CAM and lectures on related issues on distributed dynamics were given at the National Research Council, Rome, Italy; the Institute for Information Processing, Pisa, Italy; the Physics Laboratories of the University of Rome, Italy; Los Alamos Laboratories; Boston University; the MIT EECS Centennial Celebration; and to many visitors here at the Laboratory for Computer Science.

A talk on the state of the art on "Ballistic Computation" was given by Tommaso Toffoli at the 1983 Convention of the American Physical Society; and one on Digital Information Mechanics was given by Ed Fredkin at the 1983 meeting of the Jasons in Washington, DC.

Norman Margolus spent the fall term at California Institute of Technology on invitation by the physicist Stephen Wolfram, and then one more month at California Institute of Technology assisting Richard Feynman in his lectures on Information Mechanics.

2. SEMI-INTELLIGENT CONTROL

Work on semi-intelligent control has proceeded quite slowly this year, owing to our many other engagements. Most of the work was maintenance, adapting the control of the hinge system to the Lisp Machine, and student training.

Publications

1. Margolus, N. H., Toffoli, T. and Vichniac, G. Y. "Experimental Discrete Mathematics With a Fast Cellular-Automaton Simulator," SIAM Conference on the Applications of Discrete Methods, MIT, Cambridge, MA, June 1983.

2. Margolus, N.H. "Physics-like Models Of Computation," submitted to Physica D.

3. Toffoli, T. "Cellular Automata as an Alternative to (rather than an approximation of) Differential Equations in Modelling Physics," submitted to Physica D.

4. Toffoli, T. "A High-Performance Cellular-Automaton Machine," submitted to Physica D.

5. Vichniac, G.Y. "Simulating Physics With Cellular Automata," submitted to Physica D.

6. Vichniac, G.Y. "Instability in Discrete Algorithms and Exact Reversibility," SIAM Conference on the Applications of Discrete Methods, MIT, Cambridge, MA, June 1983.

Theses in Progress

1. Margolus, N. "Physics and Computation," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1984.

Talks

1. Fredkin, E. "Digital Information Mechanics,"
   The Jasons, Washington, DC, May 30,1983;
   The Jasons, La Jolla, CA, June 27, 1983;
   Computer Science Seminar, S.U.N.Y. at Oneonta, NY, January 1983.

2. Toffoli, T. "Ballistic Computation," invited talk at The American Physical Society Meeting, Los Angeles, CA, March 1983.

3. Toffoli, T. Seminars on "Parallel Computation and Distributed Dynamics," Istituto per le Applicazioni del Calcolo, Consiglio Nazionale delle Ricerche, Roma, Italy, December 1982 (and sequels at various other institutions in Italy).

4. Vichniac, G. and Toffoli, T. "Simulating Physics With Cellular Automata," Physics Seminar, Boston University, Boston, MA, May 27, 1983.

PRECEDING PAGE BLANK-NOT FILMED

PROGRAMMING METHODOLOGY

Academic Staff

B. H. Liskov, Group Leader

Research Staff

P. R. Johnson	R. W. Scheifler

Graduate Students

T. Bloom	J. N. Lancaster
S.-Y. Chiu	G. T. Leavens
P. W. Dirks	B. M. Oki
L. Gilbert	J. P. Restivo
C. E. Henderson	E. F. Walker
M. P. Herlihy	W. E. Weihl

Undergraduate Students

E. L. Raible	R. Schooler

Support Staff

A. Rubin

1. INTRODUCTION

The Programming Methodology Group has continued to study issues in distributed computing. Our focus has been on the Argus programming language and system, which supports the construction and execution of distributed programs. An overview of Argus is given in [14], and a rationale for some of our design decisions appears in [15].

During the current year, we have completed the design of a preliminary version of Argus. and have begun work on a first implementation. The implementation is being done on Vaxes running Berkeley Unix and connected via an LCS net. The first stage of the implementation, consisting of moving CLU to the Vax, was completed at the end of last summer. The second stage, the implementation of the Argus compiler, is nearly completed, and we are now working on the Argus runtime support.

Below, we give a brief overview of Argus and then discuss some topics in the Argus design and implementation. Section 3 discusses how users can implement new types of atomic objects in Argus. Section 4 discusses a way of organizing stable storage to permit efficient recovery of Argus guardians. The final section discusses dynamic modification of programs.

2. OVERVIEW OF ARGUS

Argus is a language and system that supports distributed programs, i.e., programs that run on a distributed hardware base. Each node in this base is an independent computer consisting of one or more processors and some local memory; the nodes may differ in the number and types of processors, the amount of memory, and in the attached I/O devices. The nodes can communicate only by sending messages over the network; we make no assumptions about the network topology, except that every node can communicate with every other node.

In Argus, an application is implemented from one or more modules called *guardians*. Each guardian consists of some data objects and some processes to manipulate those objects. A guardian can be thought of as "owning" the objects that it contains; each object belongs to exactly one guardian. The processes within a guardian can share the objects directly, but sharing of objects between guardians is not permitted. Instead a guardian provides access to its objects via a set of operations called *handlers* that can be called from other guardians. Arguments to handler calls are passed by value; it is impossible to pass a reference to an object in a handler call. This rule ensures that all references to an object are within that object's guardian.

Each guardian resides at a single physical node, although a node may support several guardians. Guardians survive crashes of their node of residence and other hardware failures with high probability, and are therefore *resilient*. When a guardian's node crashes, all processes within the guardian are lost, but a subset of the guardian's objects, referred to as the guardian's *stable state*, survives. After a crash, the guardian recovers with its stable s'ate intact; it then runs a special recovery process to recover the remainder of its objects. Resilience is accomplished by copying the guardian's stable objects to stable storage [11] periodically.

In addition to guardians, Argus also provides atomic actions [4], [5], [6] and atomic data types. Actions are the primary method of carrying out computations in Argus. Actions terminate by either committing or aborting, and are serializable [6], [17] and recoverable provided that the only data shared among them is atomic. An action starts at one guardian but can spread to other guardians by means of handler calls. When an action completes it either commits at all guardians or aborts at all guardians. The Argus implementation uses a two-phase commitment protocol [7], [11] to ensure this latter property.

Argus provides special types, called atomic types, that ensure proper synchronization and recovery for the actions that use them. There are a number of built-in atomic data types, for example, atomic arrays and atomic records. For atomic arrays and the other built-in atomic types, operations are classified as readers and writers, and readers and writers exclude one another in the usual way.

Serializability for atomic arrays and the other built-in types is implemented using strict two-phase locking [6] with read and write locks. The locks are acquired automatically when a primitive operation (like *fetch* or *addh*) is called by an action, and are held until the calling action terminates (commits or aborts). Recoverability is implemented by making a copy, called a *version*, the first time an action executes a writer operation. All changes are made to this copy. The copy replaces the original if the action commits; if the action aborts the copy is discarded. If an action modifies a stable object, the new value of the object is copied to stable storage as part of the commitment protocol for the action, as discussed in Section 4 below.

3. USER-DEFINED ATOMIC TYPES

In addition to the built-in atomic types, Argus provides primitives to support the implementation of new user-defined atomic types. In this section, we discuss how user-defined atomic types can be implemented in Argus.

An atomic data type, like an ordinary abstract data type [13], provides a set of objects and a set of operations. As with ordinary abstract types, the operations provided by an atomic type are the only way to access or manipulate the objects of the type. Unlike regular types, however, an atomic type provides serializability and recoverability for actions that use objects of the type. For example, relations in most relational databases provide operations to add and delete tuples, and to test for the existence of tuples; these operations are synchronized (for example, using two-phase locking [6]) and recovered (for example, using logs [8]) to ensure the atomicity of actions using the relations.

In addition to providing synchronization and recovery for using actions, atomic objects must also be resilient. We do not discuss how resilience is implemented in Argus; details can be found in [19].

In writing specifications for atomic types, we have found it helpful to pin down the behavior of the operations, initially assuming no concurrency and no failures, and to deal with concurrency and failures later. In other words, we imagine that the objects will exist in an environment in which all actions are executed sequentially, and in which actions never abort. This approach is particularly useful in reasoning about an action that uses atomic objects. The atomicity of actions means that they are "interference-free," so we can reason about the partial correctness of an individual action without considering the other actions that might be sharing objects with it [1]. This reasoning process is essentially the same as for sequential programs; the only information required about objects is how they behave in a sequential environment.

A description of a type's behavior in the absence of concurrency and failures is only part of the type's specification, however. It is also necessary to describe the kinds of concurrent executions permitted by the type and how the type handles failures. For the type to be atomic, these executions must be constrained so that actions using objects of the type are serializable and recoverable.

Atomicity of actions is a global property, because it is a property of all of the actions in a system. However, atomicity for a type must be a local property: It deals only with the events (invocations and returns of operations and commits and aborts of actions) involving the particular type. Such locality is essential if atomic types are to be specified and implemented independently of each other and of the actions that use them.

The local atomicity property used in Argus is *dynamic atomicity*; it characterizes the behavior of a class of types that ensure serializability dynamically based on the order in which actions execute operations provided by the type.

Dynamic atomicity defines limits on the concurrency that can be permitted by an atomic type. These limits can be stated informally as follows: If the sequences of operations executed by two concurrent actions on a type conflict, then some operation executed by one of the actions must be delayed until the other has completed (i.e., committed or aborted). Two actions are said to conflict if one has observed the effects of the other, or if one has invalidated the results of the other's operations.

For example, consider an atomic array of integers *a*, and suppose that an action A performs *store*(a, 3, 7), and then a second action B calls *fetch*(a, 3). If B receives the result "7" from this invocation, then B has observed the effects of A, and A must be serialized before B. If A has not yet committed when B observes its effects, A and B could exchange roles and execute operations on a second atomic array (B executing *store* and then A executing *fetch*), giving the conflicting constraint that B must be serialized before A. The resulting execution is not serializable. Since A and B conflict (B observes the effects of A), dynamic atomicity requires B's call of *fetch* to be delayed until A has completed, thus preventing this kind of non-serializable behavior.

As another example, suppose an action A executes the *size* operation on an atomic array object, receiving *n* as the result. Now suppose another action B executes *addh*. (Arrays in Argus are dynamic; the *addh* operation makes the array grow by one element.) The *addh* operation changes the size of the array, so B has invalidated the results of an operation executed by A. Thus, A must be serialized before B. As in the previous example, if A has not yet completed when B executes *addh*, A and B could exchange roles at another object, resulting in non-serializable behavior. Since A and B conflict (B invalidates the results of an operation executed by A), dynamic atomicity prevents this situation from arising by requiring B's call of *addh* to be delayed until A has completed.

Dynamic atomicity defines the limits on concurrency that an atomic type can permit. However, the implementation of an atomic type need not allow all of the concurrency permitted by dynamic atomicity. Indeed, it is often too expensive and too complicated to provide all of the permissible concurrency. For example, the implementation of atomic arrays in Argus prohibits two stores to different indexes from proceeding in parallel, even though the calls do not conflict.

3.1. Implementing User-Defined Atomic Types in Argus

Users may very well define new atomic types that permit a great deal of concurrency. Although an implementation of an atomic type need not allow all of the concurrency permitted by the type's specification, for performance reasons it may be desirable to allow much of the permitted concurrency. If users were constrained to implementing new atomic types only in terms of the built-in atomic types, the desired concurrency could not be achieved, since the built-in atomic types in Argus are limited in their provision of concurrency.

To understand the problems arising from constraining users to implement new atomic types only in terms of the built-in atomic types, consider the *semi_queue* data type. Semi_queues are similar to queues except that dequeuing happens in a non-deterministic rather than a strict FIFO order. They have three operations: *create*, which creates a new, empty semi_queue; *enq*, which adds an element to a semi_queue; and *deq*, which removes and returns an arbitrary element that was enqueued previously and has not yet been dequeued.

Semi_queues have very weak concurrency constraints. Two *enq* operations do not conflict with each other, nor do an *enq* and a *deq* operation or two *deq* operations as long as they involve different elements. Thus many different actions can *enq* concurrently, or *deq* concurrently. Furthermore one action can *enq* while another *deq*'s provided only that the latter not return the newly *enq*'d element.

We impose the following liveness requirement on semi_queues: *deq* must eventually remove any element *e* that is eligible for dequeuing. The semi_queue could be used in a printer subsystem, in which actions submit files to be printed, and the subsystem prints a file once the action that submitted it has committed. The liveness constraint on the element returned by *deq* is enough for the printer subsystem to guarantee that each file submitted by an action that later commits will eventually be printed.[6] The semi_queue data type could be implemented using an atomic array as a representation, e.g.,

rep = atomic_array[elem].

In this case, the implementation of *enq* would simply be to *addh* the new element to the atomic array. Since *addh* is a writer, an *enq* operation performed on behalf of some action A would exclude *enq* and *deq* operations from being performed on behalf of other actions until A completed. As observed above, the specification of the semi_queue permits much more concurrency than this. Note that the potential loss of concurrency is substantial since actions can last a long time. For example, an action that performed an *enq* may do a lot of other things (to other objects at other guardians) before committing.

To avoid loss of concurrency, users need a way to implement new atomic types directly from non-atomic types. In the remainder of this section we describe how user-defined atomic types can be implemented in Argus. To some extent, implementing an atomic type is similar to implementing other abstract types. The implementation must define a representation for the atomic objects, and an implementation for each operation of the type in terms of that representation. However, the implementation of an atomic type must solve some problems that do not occur for ordinary types, namely: inter-action synchronization, making visible to other actions the effects of committed actions, and hiding the effects of aborted actions.

[6]This is not quite true when we consider failures: the action that dequeues a file to print it could abort every time, preventing any progress from being made. As long as failures do not occur sufficiently often to cause this situation, every file will be printed eventually.

A way of thinking about the above set of problems is in terms of events that are of interest to an implementation of an atomic type. Like implementations of regular types, these implementations are concerned with the events corresponding to operation calls and returns; here, as usual, control passes to and from the type's implementation. In addition, however, events corresponding to termination (commit and abort) of actions that had performed operations on an object of the type are of interest to the type's implementation.

In our approach, implementations of user-defined atomic types are not informed about commit and abort events, but must instead find out about them after the fact through the use of objects of built-in atomic types. The representation of a user-defined atomic type will therefore be a combination of atomic and non-atomic objects, with the non-atomic objects used to hold information that can be accessed by concurrent actions, and the atomic objects containing information that allows the non-atomic data to be interpreted properly. The built-in atomic objects can be used to ask the following question: Did the action that caused a particular change to the representation commit (so the new information is now available to other actions), or did it abort (so the change should be forgotten), or is it still active (so the information cannot be released yet)? The operations available on built-in atomic objects have been extended to support this type of use, as will be illustrated below.

The use of atomic objects permits operation implementations to discover what happened to previous actions and to synchronize concurrent actions. However, the implementations also need to synchronize concurrent operation executions. Here we are concerned with *process concurrency* (as opposed to action concurrency), i.e., two or more processes are executing operations on the same object at the same time.

We provide process synchronization by means of a new data type called *mutex*. Mutex objects provide mutual exclusion, as implied by their name. A mutex object is essentially a container for another object. This other object can be of any type, and mutex is parameterized by this type. An example is

mutex[array[int]]

where the mutex object contains an array of integers. Mutex objects are created by calling operation

create = **proc** (x: T) **returns** (**mutex** [T])

which constructs a new mutex object containing *x* as its value. The contained object can be retrieved later via operation

get_value = **proc** (m: **mutex** [T]) **returns** (T)

This operation delivers the value of the mutex object, namely (a pointer to) the contained T object, which can then be used via T operations. *Get_value* can be called via the syntactic sugar *m.value* where *m* is a mutex object.

The **seize** statement is used to obtain exclusive use of a mutex object:

seize *expr* **do** *body* **end**

Here *expr* must evaluate to a mutex object. If that object is not now in the possession of a process, the process executing the **seize** statement *gains possession* and executes the *body*. Possession is *released* when control leaves the *body*. If some process has possession, this process waits until possession is released.[7]. If several processes are waiting, one is selected fairly as the next one to gain possession.

Inside the body of the **seize** statement it is possible to release possession temporarily by executing the **pause** statement:

pause

Execution of this statement releases possession of the mutex object that was obtained in the smallest statically containing **seize** statement. The process then waits for a system determined amount of time, after which it attempts to regain possession; any competition at this point is resolved fairly. Finally, once it gains possession it starts executing in the *body* at the statement following the **pause**.

The combination of **seize** with **pause** gives a structure that is similar to monitors [10]. For example, the use of invariants is similar in the two mechanisms: An invariant can be assumed to hold at the beginning of the body of the **seize** and after a **pause**; the code must ensure that the invariant holds at the end of the body, and before executing **pause**. However, **pause** is simply a delay; there is no guarantee that when the waiting process regains possession, the condition it is waiting for will be true.[8]The combination of **seize** and **pause** also differs from monitors in that the programmer has no control over scheduling of processes. The reason why we do not provide an analog of a monitor's condition variables is the following: Often the conditions processes are waiting for concern commit and abort events. These are not events over which other user processes in **seize** statements have any control. Therefore, it would not make sense to expect user processes to signal such information to each other.

[7]A runtime check is made to see if possession is held by this process. In this case, the **seize** statement fails with the exception *failure* ("deadlock")

[8]In Mesa [12] there is similarly no guarantee when a waiting process awakens.

3.2. Implementation of Semi_queues

In this section we present an example implementation of the semi_queue data type described earlier. For simplicity we are assuming the elements in the semi_queue are integers. We use this example to illustrate how objects of built-in atomic types can be used to find out about the completion of actions, and how mutex can be used to synchronize user processes.

The plan of this implementation is to keep the enqueued integers in a regular (non-atomic) array. This array can be used by concurrent actions, but it is enclosed in a mutex object to ensure proper process synchronization. Any modification or reading of the array occurs inside a **seize** statement on this containing mutex object.

To determine the status of each integer in the array, we associate with each integer an atomic object that tells the status of the action that inserted or deleted that item. For this purpose we use the built-in atomic type, *atomic_variant*. Atomic variant objects are similar to variant records. An atomic variant object can be in one of a number of states; each state is identified by a tag and has an associated value. Thus we represent semi_queues by objects of type

```
mutex[buffer]
```

where

```
buffer = array[qitem]
qitem = atomic_variant[present: int, absent: null].
```

If the qitem is in the "present" state, then the associated value is the enqueued integer. In the "absent" state, the value of the qitem does not matter, so we use type *null*, which has a single object, *nil*.

Atomic_variants provide operations to create new objects, and to read and modify existing objects. For each tag *t*, operation *make_t* creates a new variant object in the *t* state; this state is the object's "base" state, and the object will continue to exist in this state even if the creating action aborts. For example,

```
qitem$make_absent(nil)
```

creates a new *qitem* object in the "absent" state. Operation *change_t* changes the tag and the value of the object; this change will be undone if the calling action aborts. For example,

```
qitem$change_present(q, 3)
```

will cause *q* to be in the "present" state with associated value 3 (provided the calling action commits). There are also operations to decompose atomic variant objects, although these are usually called implicitly via special statements. Atomic variant operations are classified as readers and writers; for example, *change_t* is a writer, while *make_t* is a reader.

In this example, atomic variant objects will be decomposed using the **tagtest** statement.

```
tagtest expr
   { tagarm }
   [ others : body ]
   end
```

where

```
tagarm ::= tagtype idn [ (decl) ] : body
tagtype ::= tag | wtag
```

(In the syntax, optional clauses are enclosed with [ ], zero or more repetitions are indicated with { }, and alternatives are separated by |.) The *expr* must evaluate to an atomic variant object. Each *tagarm* lists one of the possible tags; a tag can appear on at most one arm. An arm will be selected if the atomic variant object has the listed tag, and the executing action can obtain the object in the desired mode: read mode for **tag** and write mode for **wtag**. If an arm can be selected, the object is obtained in the desired mode. Then, if the optional declaration is present, the current value of the atomic variant object is assigned to the new variable. Finally, the associated *body* is executed. If no arm can be selected and the optional **others** arm is present, the *body* of the **others** arm is executed; if the **others** arm is not present, control falls through to the next statement.

An Argus cluster implementing semi_queue appears in Figure 12.1. (The keyword **cvt** in the interface of an operation indicates that the given argument or result object is viewed as an object of the representation type inside the cluster, and as an object of the type defined by the cluster outside the cluster.) The semi_queue operations are implemented as follows. The *create* operation simply creates a new empty array and places it inside of a new mutex object. The *enq* operation associates a new atomic variant object with the incoming integer; this variant object will have tag "present" if the calling action commits later, and tag "absent" if it aborts. Then *enq* seizes the mutex and adds the new item to the contained array.

The *deq* operation seizes the mutex and then searches the array for an item it can dequeue. If an item is enqueued and the action that called *deq* can obtain it in write mode, that item is selected and returned after changing its status to "absent". Otherwise the search is continued. If no suitable item is found, **pause** is executed and later the search is done again.

Proper synchronization of actions using a semi_queue is achieved by using the

Figure 8-1: Implementation of the Semi_queue Type

```
semi_queue = cluster is create, enq, deq

qitem = atomic_variant[present: int, absent: null]
buffer = array[qitem]
rep = mutex[buffer]

create = proc () returns (cvt)
  return(rep$create(buffer$new()))
  end create

enq = proc (q: cvt, i: int)
  item: qitem := qitem$make_absent(nil)    % "absent" if action aborts
  qitem$change_present(item, i)        % "present" if action commits
  seize q do
    buffer$addh(q.value, item)         % add new item to buffer
    end
  rep$changed(q) % notify system of modification to buffer
                 % used for implementing resilience
  end enq

deq = proc (q: cvt) returns (int)
  cleanup(q)
  seize q do
    while true do
      % look at all items in the buffer
      for item: qitem in buffer$elements(q.value) do
          tagtest item % see if item can be dequeued by this action
            wtag present (i: int): qitem$change_absent(item, nil)
                         return(i)
            end % tagtest
        end % for
      pause
      end % while
    end % seize
  end deq
```

```
cleanup = proc (q: rep)
  enter topaction      % start an independent action
    seize q do
      b: buffer := q.value
      for item: qitem in buffer$elements(b) do
        tagtest item
          tag absent: buffer$reml(b) % remove only qitems in the "absent" state
          others: return
          end % tagtest
        end % for
      end % seize
    end % enter -- commit cleanup action here
  end cleanup

end semi_queue
```

qitems in the buffer. An *enq* operation need not wait for any other action to complete. It simply creates a new *qitem* and adds it to the array. Of course, it may have to wait for another operation to release the mutex object before adding the *qitem* to the array, but this delay should be relatively short. A *deq* operation must wait until some *enq* operation has committed; thus it searches for a *qitem* with tag "present" that it can write.

The *qitems* are also used to achieve proper recovery for actions using a semi_queue. Since the array in the mutex is not atomic, changes to the array made by actions that abort later are not undone. This means that a *deq* operation cannot simply remove a *qitem* from the array, since this change could not be undone if the calling action aborted later. Instead, a *deq* operation changes the state of a *qitem*; the atomicity of *qitems* ensures proper recovery for this modification. If the action that called deq aborts, the qitem will revert to the "present" state; if that action commits, the *qitem* will have tag "absent" permanently.

Qitems that are permanently in the "absent" state are also generated by *enq* operations called by actions that abort later. They have no effect on the abstract state of the semi_queue, but leaving them in the array wastes storage, so the internal procedure *cleanup*, called by *deq*, removes them from the low end of the array. (A more realistic implementation would call *cleanup* only occasionally.) It seems characteristic of the general approach used here that reps need to be garbage collected in this fashion periodically.

Cleanup cannot run in the calling action because then its view of what the semi_queue contained would not be accurate. For example, if the calling action had previously executed a *deq* operation, that *deq* appears to have really happened to a later operation execution by this action. But of course the *deq* really hasn't happened, because the calling action has not yet committed.

To get a true view of the state of the semi_queue, *cleanup* runs as an independent action. This action has its own view of the semi_queue, and since it hasn't done anything to the semi_queue previously, it cannot obtain false information. The independent action is started by the **enter** statement:

enter topaction *body* **end**

It commits when execution of the *body* is finished.

An independent action like the *cleanup* action commits while its calling action is still active. Later the calling action may abort. Therefore, the independent action must not make any modifications that could reveal intermediate states of the calling action to later actions. The *cleanup* action satisfies this condition because it performs a *benevolent side effect*: a modification to the semi_queue object that cannot be observed by its users.

4. ORGANIZATION OF STABLE STORAGE

As mentioned in Section 2, guardians are resilient to crashes. Each guardian contains a number of variables that define its state. Some of these variables may be declared to be *stable*. The objects accessible from the stable variables are referred to as *stable objects*; collectively the stable objects constitute a guardian's *stable state*. The remainder of the guardian's objects are volatile. A guardian's stable state is written periodically to stable storage devices [11]; these are devices that, with very high probability, retain the information recorded on them in spite of node and media failures.

When a node crashes, the volatile state of guardians running at that node is lost, but not the stable state. After a node crash, the Argus system restarts each guardian at the node and restores its stable state from the stable storage devices. Then the guardian can run some user-defined code to restore its volatile state to be consistent with its stable state. After the volatile state has been restored, the guardian can continue processing where it left off before the crash.

To ensure consistent data at many different guardians, writing to stable storage occurs when atomic actions commit. Before an action can commit, all changes it made to stable objects must be recorded on stable storage devices. The Argus system uses a standard two-phase commit protocol [7], [11] to ensure that an action either commits everywhere or aborts everywhere; stable objects modified by that action are copied to stable storage during the prepare phase of two-phase commit.

During the current year, we have investigated a method of implementing stable storage for guardians. Oki [16] designed a recovery system that assumes the existence of stable storage devices and organizes that storage to facilitate guardian recovery; this system is discussed further below. A stable storage device abstraction designed to match the needs of the recovery system was designed by Raible [18].

4.1. The Recovery System

In Argus, an action may visit many guardians while it executes, modifying objects at some or all of these guardians. One simple method of implementing recovery is to record the entire stable state of each visited guardian during the prepare phase for the committing action. However, an action will usually modify only a small proportion of the stable objects at a guardian; writing all objects to stable storage involves lots of unnecessary work. Therefore, we have designed an alternative technique.

In Argus guardians, all stable objects are atomic objects as discussed in the preceding section. These objects may be either of built-in types or user-defined types. The following discussion considers only the built-in types; for a discussion of user-defined types see [16].

While an action is executing at a guardian, the Argus system keeps track of all atomic objects at that guardian that the action modified. This information is kept in the *Modified Objects Set* or *MOS*. A separate MOS is kept for the action at each guardian it visits.

Every atomic object has a *uid*, a name that is unique with respect to its guardian. This name is used as a reference on stable storage; one atomic object can refer to another by containing its uid. Having such a name permits us to copy one atomic object to stable storage without having to copy all the other atomic objects it refers to at the same time, since we can represent these references by the uids of the referenced objects.

The recovery system maintains an *Accessibility Set* or *AS*. This set keeps track of which atomic objects have copies on stable storage. The AS is guaranteed to contain all objects that are accessible from the stable variables. In addition, it may contain some objects that were accessible previously but are no longer accessible. These additional objects are removed from the AS periodically when the Argus system does garbage collection.

When an action commits, the recovery system must copy all objects to stable storage that are listed in both the MOS and the AS. In addition, it must copy any objects that are *newly accessible*: An object becomes newly accessible when an existing accessible object is modified to refer to it. Usually newly accessible objects are also newly created by the action that made them accessible, but not always; such objects may have existed previously in the volatile state of the guardian.

In our scheme, stable storage is organized as a *log* containing *entries*. It is possible to read any entry in the log, but new entries can only be appended to the top of the log. Entries are copied to the log in one of two modes: either they are *written* or they are *forced*. Forcing an entry to the log guarantees that the entry (and any previous unforced entries) has actually been recorded on stable storage by the time the operation returns. By contrast, when an entry is written it may not have been recorded on stable storage by the time the operation returns.

Entries on the log come in two varieties: *data entries* and *outcome entries*. Data entries simply record the state of some atomic object. Outcome entries record the outcomes of actions, e.g., prepared, committed. All outcome entries are chained together; the most recently written outcome entry points to the outcome entry written second most recently and so on. Certain outcome entries such as prepared entries also contain pointers to a number of data entries.

Now we can describe how the recovery system carries out a prepare for some action. It makes use of two internal data structures in doing prepare: The *prepare list* keeps track of the uids of all objects copied to the log for the preparing action and the *newly accessible objects set* or NAOS keeps track of all objects found to be newly accessible while doing the prepare. Initially both these data structures are empty.

The recovery system goes through a three step process in doing prepare: First it copies all objects that are listed in both the MOS and the AS. Next it copies any objects found to be newly accessible, and adds the uids of these objects to the AS. In both of these steps it keeps track of the objects written to the log in the prepare list. Finally it writes a prepare record containing the prepare list to the log; once this is accomplished the action is prepared at this guardian. We assume in the following that only one prepare can be going on at a guardian at a time.

In the first step, the recovery system examines each object in the MOS. If that object is not listed in the AS, it is discarded. Otherwise, the *current* version of the object is copied to the log in a data entry; this current version contains the changes made by the committing action. (Such an object also has a *base* version, which records its state before the action ran; two versions are needed in case the action aborts.) In addition, the pair <uid of copied object, address of data entry> is added to the prepare list.

In copying an object, all contained non-atomic objects are copied, but contained atomic objects are not copied; they will be copied separately if necessary. (The copy in the log will contain the uids of these contained objects.) However, it is necessary to determine whether such contained objects are newly accessible, so contained objects are looked up in the AS, and, if they are not present, are added to the NAOS.

When all objects in the MOS have been examined, step 1 is finished. In step 2, each object in the NAOS is copied to the log. First, the newly accessible object is removed from the NAOS and added to the AS. Then the base version of the object is copied in a special *base committed* outcome entry: this special entry is needed in case the object has also been made accessible by some other action that has not yet prepared. Using a special entry guarantees that the object will be restored after a crash if either of the actions commits; see [16] for more details.

Next, the recovery system checks whether the newly accessible object has a current version in addition to a base version. There are three possibilities here; either the preparing action modified the object, or the object was modified by some action that has already prepared (but has not committed or aborted), or the object was modified by some action that is still active. If the preparing action modified the object, the current version of the object is copied in a data entry, and the <object uid, data entry address> pair is added to the prepare list. If the object was modified by an action that has already prepared, its current version is forced to the log in a special *prepared data* outcome entry. This entry should be thought of as a extra piece of the prepare information already stored in the log for the prepared action. If the object was modified by an active action, nothing further need be done; the current version will be copied to the log if the active action prepares.

In copying either the base or current version, if any newly accessible objects are discovered they are added to the NAOS.

When the NAOS is empty, step 2 is finished. Then the recovery system forces a prepared entry to the log. The entry contains the name of the preparing action and the prepare list and also a link to the previous outcome entry.

In addition to the prepared entry, there are a number of other outcome entries: committed, aborted, committing and done entries. These are all forced to the log at the appropriate time. For example, the committing entry is forced to the log by the coordinator just before it enters phase two of two-phase commit; this entry contains the action id of the committing action and the identities of all the guardians that are participants in the commit.

After a crash, the recovery system restores the guardian state by processing the log backward, starting with the *last outcome entry forced to the log before the crash* and following the outcome entry chain. The information stored in the log is sufficient for restoring each object. For example, suppose a prepared entry for action A lists object X in the prepare list. Then there are the following possibilities:

1) An aborted entry has already been processed for A. (Notice that this entry would occur later in the log than the prepared entry for A, and therefore would be processed before the prepared entry.) In this case the entire prepared entry is ignored.

2) A committed entry has been processed for A. In this case, the recovery system checks whether X has already been restored; this would happen if some other action B that prepared and committed later than A had modified X. If X has already been restored, then its data entry is ignored; otherwise its data entry is used to restore the base version of X.

3) Neither a committed entry nor an aborted entry has been processed for A. In this case the data entry for X is used to restore the current version (not the base version) of X, and A is granted a write lock on X. In addition, A is listed as a participant in a not-yet-completed two-phase commit.

As time goes by, more and more of the information in the log becomes uninteresting. For example, all information stored on behalf of an action that has aborted is no longer needed. Neither is information about a modification to an object that has been modified subsequently by an action that committed. To speed up restoring, housekeeping must be done to the log periodically to remove uninteresting information. Oki investigated two methods of housekeeping. Each involved building a new log that contained only interesting information. One scheme builds the new log by processing the information in the old log in a fashion very similar to what is done when restoring the guardian. The other scheme takes a snapshot of the guardian by starting at the stable variables and making copies of all accessible objects. Details of the two schemes can be found in [16]; our analysis indicates that the snapshot scheme is the more efficient of the two.

The method described above is a hybrid of a simple logging technique with a shadowing scheme. In a simple logging technique [2], [7], [11], a prepared entry simply records the preparing action, but does not list the objects copied in preparing the action. Instead, the entry made when copying an object lists the action on whose behalf the copy was made. Such a scheme avoids the need to build the prepare list, and also has smaller prepared entries than our scheme. However, recovery is slower than in our scheme, since every entry in the log must be examined during recovery.

In a shadowing scheme [7], the system maintains a *map* that points to the current version of each object. When preparing an action, a new copy of the map in which the entry for each modified object points to the new version is written to stable storage. A pointer to this new map is stored in a log as part of the prepared entry for the action. Later, when the action commits, the new map replaces the old one.

Shadowing is fast for recovery, since exact information is kept about the current versions of objects. However, it is slow during prepare if the maps are large.

In our scheme, we actually keep the map, but we distribute it among the prepared entries. Our scheme should be as fast as simple logging during prepare. It should be substantially more efficient than logging during recovery, although not as efficient as a shadow scheme. We believe that it can do restoring reasonably fast when combined with the snapshot housekeeping scheme described above.

5. DYNAMIC MODIFICATION OF PROGRAMS

Programs in Argus will commonly take the form of *subsystems* that provide services to other programs called *clients*. Examples of subsystems are mail and naming services. Typically such subsystems will provide a logically centralized service in a distributed manner. For example, to its users the mail subsystem appears to be a single entity that delivers mail to mailboxes of users; however, these mailboxes may be distributed on many nodes of the network.

If subsystems remain in use for a reasonable length of time, they will probably need to be modified. Modifications arise for many different reasons. For example, a subsystem may be modified to fix an error in its implementation, or to change its implementation to one that is more efficient in some dimension. Or, a subsystem may be modified to provide more services than it did previously. Thus, long-lived subsystems evolve over time.

The services provided by many subsystems include storing information for their clients. For example, a mail subsystem stores messages for its users, while a naming subsystem stores information about how to resolve higher-level names to lower-level names. When such subsystems evolve, it is important that they not lose information entrusted to them.

Bloom [3] has studied the problem of supporting subsystem evolution without loss of information and with minimal disruption of client programs. In particular, bringing an entire system down in order to replace a subsystem with a newer version was ruled out as being too disruptive. A special problem in a distributed system is that it is difficult to find a time to schedule a system shutdown, since a good time at one location is likely to be a bad time at another. Even in centralized systems, where it may be possible to find a good time for a system shutdown, supporting subsystem evolution by shutting down the system is not as satisfactory as a method that does not involve shutdown. Bloom developed a method of doing subsystem replacement dynamically, while all other parts of the system continue to run. In particular, the replacement is invisible to clients; they may notice that the subsystem cannot be used for a while, but that is the only effect of the replacement.

Bloom studied subsystem replacement in the context of Argus. A subsystem was thought of as providing an interface consisting of a collection of handlers that clients could call. Thus, to its clients, a subsystem appears to be a guardian. However, a subsystem is typically implemented by many guardians residing at many nodes in the network.

The first question that must be addressed when doing subsystem replacement is: when should it be considered *permissible* for a new subsystem to replace an existing one? Bloom's primary criterion was that the new subsystem's behavior not be different from the old one's in any way detectable to clients. If the services provided are going to change, then clients cannot simply be hooked up to the replacement, since their behavior may also change. Instead some human intervention will be needed in such a case.

It may seem that a replacement is permissible only if the interface of the subsystem does not change, i.e., it provides the same handlers, taking the same types of arguments and returning the same types of results, and having the same specifications. However, a more general definition is possible. One generalization is that a replacement is permissible if it provides an *extension* of the previously available services. This means the replacement must contain all the old handlers with the same types of arguments and results, but in addition it can provide new handlers. Furthermore the behavior of the old handlers must still be the same as it was in the replaced system. (The clients using the old system see the same behavior they always did; new clients see the extended specification.) Thus the question about permissibility can be posed as a question about what theorems can be proved from a program's specification. If every theorem that could be proved about the old subsystem can still be proved about the new subsystem, then the replacement is permissible.

For example, consider a mail system that provided no operation for removing users and suppose that a replacement adding such an operation was desired. This replacement would be permissible only if the following theorem could *not* be proved of the original system:

> If u is entered as a user of the mail subsystem, then u will be a user of the mail subsystem from then on.

The requirements for permissible replacement can be relaxed in other ways as well. One example is a new subsystem designed specifically to replace an existing one. The new subsystem need not meet the full specification of the original subsystem, since it will never start from an initial state. It need meet only those parts of the specification that relate to states reachable from the last state of the replaced subsystem. If we consider non-deterministic specifications of subsystems, replacement requirements may be relaxed still further: the new subsystem need not behave exactly as the old one would have, since each could make different decisions when a choice was not fully determined. For complete descriptions of these eased requirements see [3].

Next, Bloom defined a method of performing a permissible replacement. This method relies heavily on the semantics of Argus. Most important are atomic actions and the fact that guardians can survive crashes. The method works as follows:

1) Create a replacement action. All activity concerning the replacement will take place within this action.

2) Destroy all guardians in the original subsystem. Destroying a guardian in Argus crashes the guardian immediately. If the destroying action commits later, the guardian will really be destroyed at that point. Otherwise, if the destroying action aborts, the guardian simply recovers from the crash.

3) Create all the replacement guardians. These are created with their stable variables in existence but uninitialized.

4) Perform a transformation function

$$T: s_o \rightarrow s_n$$

Here s_o is the stable state of the old subsystem and s_n is the stable state of the new subsystem. This new state is stored on stable storage; when T is complete, the new guardians appear as if they had crashed. T will be discussed further below.

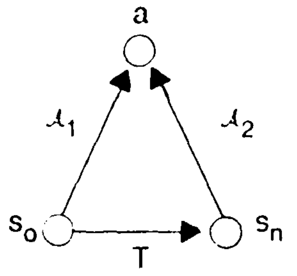
5) Bind the handlers of the old guardians to the handlers of the new guardians. The effect of binding is that when a client calls an old handler after replacement, the new handler will be called instead. Of course, binding is permitted only if the old and new handlers take the same types of arguments and return the same types of results.

6) Commit the replacement action. This causes the old guardians to be destroyed, as mentioned above. The new guardians recover from their crash.

It is important that replacement runs as an action, because this rules out any danger that a partial replacement will occur. If the action commits, then the replacement will have occurred entirely; if it aborts, then the old subsystem will continue in operation.

The correctness of a replacement depends on a number of activities performed by the user interacting with the replacement system. For example, the user must execute a correct transformation function T. T maps the current state of the old subsystem into the current state of the new subsystem. Intuitively, and ignoring a replacement that is an extension, T is correct if both the old and new subsystem states represent the same abstract state. This situation is illustrated in Figure 12-2. Here T is shown mapping the old state, s_o, into the new state, s_n. Each representation is mapped to the abstract object it represents by the abstraction function $\mathcal{A}$ [9] for its subsystem. T is correct because $\mathcal{A}_1(r) = \mathcal{A}_2(T(s_o))$. (Actually this correctness condition is too simple for subsystems with non-deterministic behavior. The exact definition is given in [3].)

Figure 12-2: DIAG Transformation Diagram



In addition to defining the transformation function, the user must identify correctly all the guardians that make up the old subsystem, create the proper number of new guardians, and bind the old handlers to new handlers correctly. Bloom discusses how the Argus system together with various naming systems can help the user with these problems. She also discusses some techniques for testing replacements in advance of actually doing them.

References

1. Best, E. and Randell, B. "A Formal Model of Atomicity in Asynchronous Systems," Acta Informatica 16, (1981), 93-124.

2. Bjork, L. A. "Generalized Audit Trail Requirements and Concepts for Data Base Applications," IBM Systems Journal 14, 3, (1975), 229-245.

3. Bloom, T. "Dynamic Module Replacement in a Distributed Programming System," MIT/LCS/TR-303, MIT Laboratory for Computer Science, Cambridge, MA, March 1983.

4. Davies, C. T. "Recovery Semantics for a DB/DC System," Proceedings of the 1973 ACM National Conference, 1973, 136-141.

5. Davies, C. T. "Data Processing Spheres of Control," IBM Systems Journal 17, 2, (1978), 179-198.

6. Eswaren, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. "The Notion of Consistency and Predicate Locks in a Database System," Communications of the ACM 19, 11 (November 1976), 624-633.

7. Gray, J. N. "Notes on Data Base Operating Systems," Lecture Notes in Computer Science 60, Goos and Hartmanis, (eds.), Springer-Verlag, Berlin, 1978, 393-481.

8. Gray, J. N., et al. "The Recovery Manager of the System R Database Manager," ACM Computing Surveys 13, 2 (June 1981), 223-242.

9. Guttag, J., Horowitz, E. and Musser, D. "Abstract Data Types and Software Validation," Communications of the ACM 21, 12 (December 1978), 1048-1064.

10. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept," Communications of the ACM 17, 10 (October 1974), 549-557.

11. Lampson, B. "Atomic transactions," Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science 105, Goos and Hartmanis, (eds.), Springer-Verlag, Berlin, 1981, 246-265.

12. Lampson, B. and Redell, D. "Experience With Processes and Monitors in Mesa," Communications of the ACM 23, 2 (February 1980), 105-117.

13. Liskov, B. and Zilles, S. N. "Programming with Abstract Data Types," Proceedings of the ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50-59.

14. Liskov, B. and Scheifler, R. W. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Computation Structures Group Memo 210-1, MIT Laboratory for Computer Science, Cambridge, MA, August 1982. Also published in ACM Transactions on Programming Languages and Systems, 5, 3 (July 1983), 381-404.

15. Liskov, B. and Herlihy, M. "Issues in Process and Communication Structure for Distributed Programs," To be published in Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems.

16. Oki, B. "Reliable Object Storage to Support Atomic Actions," MIT/LCS/TR-308, MIT Laboratory for Computer Science, Cambridge, MA, May 1983.

17. Papadimitriou, C. H. "The Serializability of Concurrent Database Updates," Journal of the ACM 26, 4 (October 1979), 631-653.

18. Raible, E. L. "A Log-based Interface to Stable Storage for the Argus Language," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

19. Weihl, W. and Liskov, B. "Specification and Implementation of Resilient, Atomic Data Types," Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, June 1983, 53-64. Also Computation Structures Group Memo 223, MIT Laboratory for Computer Science, Cambridge, MA, December 1982.

Publications

1. Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W. E. "On Reaching Approximate Agreement in the Presence of Faults," to appear.

2. Henderson, C. E. "Locating Migratory Objects in an Internet," Computation Structures Group Memo 224, MIT Laboratory for Computer Science, Cambridge, MA, January 1983.

3. Herlihy, M. and Liskov, B. "A Value Transmission Method for Abstract Data Types," ACM Transactions on Programming Languages and Systems 4, 4 (October 1982), 527-551.

4. Liskov, B. and Herlihy, M. "Issues in Process and Communication Structure for Distributed Programs," to appear.

5. Liskov, B. and Scheifler, R. W. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," To be published in ACM Transactions on Programming Languages and Systems.

6. Weihl, W. E. "Data-dependent Concurrency Control and Recovery," To be published in Proceedings of the ACM Symposium on Principles of Distributed Computing, August 1983.

7. Weihl, W. E. and Liskov, B. "Specification and Implementation of Resilient, Atomic Data Types," Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, June 1983, 53-64. Also Computation Structures Group Memo 223, MIT Laboratory for Computer Science, Cambridge, MA, December 1982.

Theses Completed

1. Bloom, T. "Dynamic Module Replacement in a Distributed Programming System," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

2. Dirks, P. W. "Distributed Computation Through an Extended Remote Procedure Call Mechanism," S.B. and S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

3. Henderson, C. E. "Locating Migratory Objects in an Internet," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1982.

4. Oki, B. M. "Reliable Object Storage to Support Atomic Actions," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

5. Raible, E. L. "A Log-based Interface to Stable Storage for the Argus Language," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

6. Schooler, R. "Argus Low-Level Naming Subsystem," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

Theses in Progress

1. Chiu, S-Y. "A Debugging System for Argus, A Guardian and Transaction Based Language for Distributed Computations," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

2. Gilbert, L. "The Task Construct in the Programming Language Ada," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

3. Herlihy, M. P. "Modular Composition of Fault-Tolerant Programs," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

4. Lancaster, J. N. "Naming in a Programming Support Environment," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1983.

5. Weihl, W. E. "A Method for the Construction of Modular, Reliable, Concurrent Systems," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1984.

Talks

1. Liskov, B. "Linguistic Support for Distributed Programs,"
   University of Maryland, College Park, MD, September 30, 1982
   Stanford. University, Stanford, CA, October 12, 1982
   IBM, San Jose, CA, October 13, 1982
   Harvard University, Cambridge, MA, December 1, 1982
   University of Massachusetts, Amherst, MA, December 13, 1982.

2. Liskov, B. "Argus: Linguistic Support for Distributed Computing,"
   IBM/MIT Meeting, Cambridge, MA, January 11, 1983
   Greater Boston SIGPLAN, Cambridge, MA, January 13, 1983
   Digital Equipment Corp., Marlboro, MA, March 1983
   University of Toronto, Canada, March 29, 1983
   Honeywell, Inc., Bloomington, MN, April 27, 1983.

3. Weihl, W. E. "Specification and Implementation of Resilient Atomic Data Types," SIGPLAN '83, San Francisco, CA, June 1983.

PROGRAMMING TECHNOLOGY

Academic Staff

A. Vezza, Group Leader

Research Staff

T. Anderson	D. Lebling
S. Berlin	J.C.R. Licklider
S. Galley	R. Myhill
L. Hawkinson	C. Reeve

Graduate Students

D. Buffo	R. Sun
P. Lim	A. Yeh

Undergraduate Students

R. Bily	B. Law
H. Baek	S. Malone
J. Dike	S. Meeks
P. Dufour	M. McEntee
M. Duke	J. Saks
D. Elrod	G. Shaw
M. Herdeg	B. So
C. Humphreys	M. Terpin
S. Kukolich	J. Wang
M. Kudisch	K. Wohl

Support Staff

A. Finn	N. Mims

1. INTRODUCTION

This is the report of the Programming Technology Group whose main effort is the development of a computer-based planning syst·-·· for use in situations in which voluminous data are available to support the planning. Some of this development is aimed at moving a large system of software development tools, centered upon the language MDL, from the DECSYSTEM-20/TOPS-20 environment in which it was initially constructed to an open set of environments including VAX-UNIX and systems based on the 68000 family of chips. Another part of the work is the development of a planning aide based on the essential ideas of VisiCalc but extended to include symbolic computation, data management, and graphic display. A third and smaller part of the work is aimed at the development of a graphical programming and monitoring system. Throughout the work of the group, and especially in the planning aid and the graphical programming parts, there is a focus on ease of use and other aspects of the computer-user interface.

The language, MDL, that is the centerpiece of the system of software development tools that is being transported to the VAX and 68000 environments is at the same time an extension of LISP and a production language that supports programming in a more conventional, more Fortran-like style. The acronym, "MIM," used in the remainder of this report, stands for "Machine Independent MDL."

2. MIM DEVELOPMENT

Work on the machine independent MDL (MIM) project over the past year has progressed in several areas:

1) Moving out of old MDL;

2) Making MIM more complete;

3) Development of the I/O system;

4) Development of a VAX/UNIX implementation;

5) Improvements of the compiler;

6) Improvement of performance;

7) Improvement of robustness.

At present, all compilers for MIM run in the old implementation of MDL on DECSYSTEM/20. Most of the work required to move the machine-independent part of the compiler and the open-compiler for the VAX into MIM has been done. When that work is finished, it will be possible to put a complete development environment, including compilers, onto the VAX. Then the open compiler for the DECSYSTEM/20 will follow.

Many of the MDL primitives that were originally omitted from MIM have been added. Bit manipulation routines, with associated open-compilers, have been developed; both a copy and a mark-sweep garbage collector have been debugged and made to run on DECSYSTEM/20 and the VAX. The entire preloaded MDL environment, plus a number of debugging packages, now exists for the DECSYSTEM/20 and VAX implementations. Other packages are transported or reimplemented as the need arises.

The major feature of old MDL that MIM currently lacks is the collection of memory-management routines: purification of static data structures, fast I/O of raw data structures, and so on. Work has begun on adding these features to MIM.

The stream I/O system is essentially complete. Most of the high-level I/O routines from old MDL have been implemented in MIM using streams, usually with compatible interfaces. In addition, compatible disk and terminal streams have been implemented for TOPS-20 and UNIX. Programs can thus do quite sophisticated terminal handling and file interactions, without regard for which system they are running on.

An advantage of the stream system is that other stream types can be added without modification to the MIM interpreter or kernel. Work has begun on a network interface, using TCP/IP. It will provide a basis for a message system.

The VAX/UNIX implementation is now compatible in all respects with the DECSYSTEM/20 implementation. As an example, Lebling's PLAID system was ported to the VAX from DECSYSTEM/20 with only one change: the removal of some code that was needed to get around a monitor bug in DECSYSTEM/20.

A significant investment of time was required to complete the VAX/UNIX implementation because of various problems encountered in UNIX. The majority of the terminal handling code that is built into the DECSYSTEM/20 monitor had to be written in MIM for the UNIX version since the UNIX philosophy is to have that code in the user job. In addition, care had to be taken to prevent MIM from leaving the terminal in a bad state when it returned to the UNIX shell. The size of the RK07 disks on the VAXes and the way in which UNIX handles swapping space also led to some problems. The RK07 disks have a 27 megabyte capacity which must be divided up between file space and swapping space. This limitation restricts the virtual memory size of a MIM on UNIX. The remote virtual disk will help relieve this limitation. UNIX software constraints also limit the size of a particular job's swapping space.

Because of those limitations, MIM generally runs on the edge of disaster, and it is nearly impossible to run another program in addition to MIM on a VAX. The MIM software had to take great pains to acquire as much memory as possible without killing itself.

The suite of MIM compilers has undergone further enhancements to improve the speed of compiled code, and to allow larger programs to be compiled. This includes optimizations to reduce the number of temporary stack locations required by compiled routines, optimization of register allocation on TOPS-20 and development of new primitives (such as dispatch tables) to simplify some common operations. Peep hole optimizers were added to the MIM compiler to clean up the compiled code.

The relative slowness of MIM as compared to MDL led, first, to the instrumentation of the DECSYSTEM/20 MIM kernel. This allowed us to find bottlenecks in the MIM interpreter and kernel. Many of the major problems were corrected, either by careful rewrites of frequently used routines, or by improvements in the kernel's structure. In addition, careful studies were done of the behavior of DECSYSTEM/20 MIM in the extended-addressing environment. This required the construction of a simulator of the DECSYSTEM/20 paging hardware, so measurements could be made. Given information about how MIM was competing with itself for pages and page table slots, we were able to improve performance significantly by changing the memory organization. However, because of inherent problems in the DECSYSTEM/20 memory management hardware and software, very large MIM programs will invariably have performance limitations

As MIM has acquired more users, a significant amount of time has been spent debugging both the MIM interpreter and the various MIM compilers. This has led to considerable improvement in the robustness of the system; it is now reasonable to expect even quite complicated programs to fail only because of bugs in their implementation, not because of bugs in the MIM system.

3. PLANNING SYSTEM

Last year's work on the planning aid system PLAID was motivated by the transporting of PLAID out of the old MDL environment and into the new one, and by the acquisition of "personal computers" (VAX-11/750s) with high-resolution graphics terminals (BBN BitGraph terminals). Notable achievements were:

1) Successful transport of PLAID into MIM/MDL on TOPS-20;

2) Successful transport of PLAID into MIM/MDL on VAX UNIX;

3) Extending the "continuous update" model to include graphics;

4) Implementation of a pointer ("mouse") adjunct to the user interface;

5) Implementation of HELP and MENU facilities;

3.1. Transporting PLAID to MIM

The major project over the last year was the transport of PLAID into MIM, the new implementation of MDL. PLAID was the first major user program transported, and it flushed out many bugs in the MDL compilers as well as in MDL itself. PLAID also is a very large program (the executable file is approximately 400 hundred pages on TOPS-20), and so the transportation of PLAID exposed several program-size-related bugs.

The version of PLAID in MIM/MDL differs from the old version in one major respect, which is that window management is considerably more transparent to the user (or programmer) because the stream-I/O system in MIM/MDL is considerably more flexible and extensible than the old "internal channel" system of the MDL.

A largely compatible MIM/MDL version of PLAID was running in January 1983. The only part of the old version still not converted is the "fast" LOAD and STORE commands for worksheets, which depend on memory management routines not yet implemented in MIM/MDL.

The version of PLAID in MIM/MDL is approximately 2 to 2.5 times slower than the version in old MDL. Speed improvements are expected during the coming year.

Once the MIM version of PLAID was working relatively robustly, transport of PLAID into the VAX/UNIX environment was begun. This was gratifyingly easy. Most of the problems were occasioned by problems with the small size of the VAX RK07 disks, and with certain features of the UNIX operating system.

The VAX/UNIX version of PLAID, running on a VAX-11/750, is about four times slower than the TOPS-20 version, but this varies from command to command, and often the perceived speed is as fast as that of the TOPS- 20 version.

3.2. "Continuous Update" Graphics

The "spreadsheet" model of data presentation and entry is widely popular. One reason for this popularity is that changes made to independent variables in a model propagate to the dependent variables and change the presentation continuously. The model and the presentation are always in synchronization, or in the process of synchronizing.

We have extended this method of presentation to graphics, and particularly, to graphs of various types displayed on a high-resolution display. In PLAID, a model may be presented in several different ways. The most common is as a "worksheet." In this mode, the familiar VisiCalc ledger-style of display is used. Another method of presentation is as a "graph." In this mode, graphs which express parts of a model (usually rows or columns of a worksheet) are displayed. The elements of the graph are elements of the worksheet, and therefore, if those elements change, the graph is changed as well. For example, a bar graph showing actual and projected sales of various items over several years would show a set of bars for each item, one bar per year. If the projection changed, the height of the bar would change at the same time.

To implement this method of presentation, every display window which contains a worksheet also contains a routine to redisplay the window, *and* a routine to do low-level update (such as changing the height of one bar in a bar graph).

The method may be extended to other forms of presentation as well. To give one example, a "set" is a display window or worksheet which shows only those worksheet items which satisfy some criterion.

3.3. Interface to a High-Resolution Display

PLAID was originally implemented to use the VT100 and similar terminals, so when we acquired high-resolution displays (BBN BitGraphs), PLAID was extended to use the functions available on these more powerful devices.

One feature of the BitGraph which makes it an interesting device to use is that while internally it is a DMA device, it must be controlled over a terminal line: there is no DMA to the host computer.

As discussed above, PLAID now has the ability to draw graphs, and to update them piecemeal as the worksheets they represent are changed.

The BitGraph terminal also supports a pointing device (a "mouse"), and facilities for using it have been added to the system. It may be used to move the worksheet cursor and to select items from menus.

3.4. Help and Menu Facilities

A new Help facility has been implemented. Each node of the ATN which describes the command structure may have a help text stored with it. In practice this "text" is actually a pointer into a disk file. A pop-up Help window is displayed when the Help key is pressed. When any other command key (such as EOL) is pressed, the window is removed. We expect to expand this facility to have better selection of which help texts to display in the future.

The Menu facility has been upgraded to allow items to be selected from the menu with a pointing device.

In the future we expect to do some work on optimal Menu and Help window placement. At present part of the contents of the current display window are overwritten and must be refreshed when the menu or help is cleared. (On a BitGraph this operation is extremely fast, on a VT100 it proceeds at the terminal's baud rate).

4. GRAPHICAL PROGRAMMING AND MONITORING OF PROGRAM BEHAVIOR

The rationale for the study of graphical approaches to programming and the understanding of software includes several components. One of them is that programming as practiced conventionally seems, to anyone accustomed to the graphs and diagrams of much science and most engineering, to be dominated abnormally by linear strings of abstract symbols. If you are told that programs are difficult to write and to understand and that software is usually in a chaotic state, and you then look at a hundred pages of program listings, you are likely to say, "No wonder." This part of the rationale says that, if a picture is worth a thousand words, here is a place to try to use pictures. Another part of the rationale is that many human products of the television age understand pictures better than linear text. Another part is that computer programs are intrinsically multidimensional and that presenting them as text artificially linearizes them. In any event, there are now several efforts -- for example, at Brown University, Computer Corporation of America, the University of Toronto, and SRI International -- to see how graphics can be used to facilitate the preparation and/or understanding of software.

Our project. which has been under way since last September, is aimed at graphical programming and graphical monitoring. It is exploring approaches that may help newcomers write and understand simple programs and at approaches that may help experienced programmers write and understand complex programs.

In the graphical programming part. the aim is to devise graphical figures -- pictograms -- to represent most of the basic constructs of the MDL language and to develop a system in which one can construct a program (MDL function) or data set by arranging and interconnecting those figures. We are not yet at the point of demonstrating such graphical programming. However, we have explored many ways of representing programming constructs, and it seems clear that we shall be able to program graphically in an extension of MDL. Moreover, it is apparent that there are some advantages to programming in that way. Programming is basically more like building than like writing. Using pictures (pictograms) to represent computational objects avoids some of the problems introduced by using names. If variables are represented by places on the display screen and identified by pictograms in those places, there is no way of having a name-collision between two variables, and there is no way of getting confused between "static scoping" and "dynamic scoping" rules for figuring out which of two or more variables is meant by a name that they have in common -- because they do not have any names. On the other hand, there are problems in graphical representation. Even though we strive for compactness, it tends to use more space than does representation by alphanumeric strings. When it comes to identifying a thousand or more different objects, we recognize the difficulty of learning to associate a pictogram with each object. For speakers of Western languages, words have an advantage in already established and exploitable meanings. Do speakers of Chinese or Japanese, or did writers of ancient hieroglyphs, have a superior approach to the representation of computer constructs?

In the graphical monitoring part of the work, we are extending the MDL interpreter to take some steps toward explaining what it is doing while it does it. In concept, the computer has an extra display screen on which it displays information about what it is doing. The main operative construct in MDL is a construct called the "function application form." When such a form is interpreted, the interpreter considers the first thing in the form to represent an operator and all the other things to represent operands that the operator should operate upon. We have modified the interpreter to do some more work whenever it encounters a function application form. It first presents some information about what it is going to do. Then it carries out the normal processing. And then, having seen the result arrived at by the normal processing, it presents some additional information. Thus far, the explanation that it gives is mainly alphanumeric, not graphical. We will be substituting graphical self-explication with non-graphical self-explication in the coming months.

In both parts of the work, we are interested in large, complex programs as well as in small, simple ones. However, the large, complex ones present more difficult problems. The main approach we are taking to handling large and complex programs takes advantage of the fact that most such programs -- and perhaps all such programs that are understandable -- are hierarchical in structure. An important part of the work, therefore, is figuring out how to represent computer constructs at several different levels of abstraction. It is one thing to develop pictograms for 50 to 200 constructs that are built into a programming language but quite another to represent at various levels of abstraction the thousands of programs that are in use and on the market today. Moreover, it is not likely that programmers will want to create graphical representations of programs in addition to creating the programs. We see in rough outline how to synthesize graphical program representations in parallel with the synthesis of programs and data sets, but we do not have a good approach to abstracting simplified representations from complex ones. That looks like a good and difficult research problem.

Publications

1. Yeh, A. "Ply: A System of Plausibility Inference with a Probabilistic Basis," MIT/LCS/TM-232, MIT Laboratory for Computer Science, Cambridge, MA, December 1982.

Theses Completed

1. Bily, R. "A Microcomputer Database Management System," S.B. thesis MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1982.

2. Saks, J. "The Subroutine Library as a Tool for Business Applications Programming," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1983.

3. Thompson, M. "Graphically Representing the Control Structure of MDL Programs," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

4. Yeh, A. "PLY: A System of Plausibility Inference with a Probabilistic Basis," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1983.

Theses in Progress

1. Buffo, D. "An Expert System for Financial Planning," M.S. thesis MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1984.

2. Sun, R. "Application Oriented Integrity Specification in a Data Model," M.S. thesis MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1983.

SYSTEMATIC PROGRAM DEVELOPMENT

Academic Staff

J. Guttag, Group Leader

Graduate Students

R. Forgaard
R. Kownacki
J. Wing
K. Yelick
J. Zachary

Support Staff

E. Pothier

Postdoctoral Fellow

P. Lescanne

1. INTRODUCTION

Over the last year the Systematic Program Development Group has worked on two related projects, the Larch Specification System and the RR Rewrite Rule Laboratory. The work on Larch has been supported primarily by DARPA, and the work on RR primarily by the NSF.

2. LARCH

2.1. Summary of Status and Plans

The Larch Project is developing tools and techniques intended to aid in putting formal specifications to productive use. Many of its premises and goals were discussed in last year's progress report and in [4]. In our work on Larch, the Systematic Program Development Group has cooperated closely with Jim Horning of Xerox PARC.

The central component of the project is a two-tiered approach to specification and a family of specification languages to support this approach. Each Larch language has a component derived from a programming language and another component common to all programming languages. We call the former interface languages, and the latter the shared language.

We use the interface languages to specify program modules. Specifications of the interface that one module presents to other modules often rely on notions specific to the programming language, e.g., its denotable values or its exception handling mechanisms. Each interface language deals with what can be observed about the behavior of programs written in its programming language. Its simplicity or complexity is a direct consequence of the simplicity or complexity of the observable state and state transformations of the programming language.

The shared language is used to specify abstractions that are independent of the programming language. These abstractions are used within specifications written in the interface languages. The role of shared language specifications is similar to that of abstract models in some other styles of specification.

We are still in the early phases of the Larch project. We have completed the design and documentation of the Larch Shared Language [5][6]. We have also designed interface languages for CLU [20] and Mesa.

A primitive checker for the shared language has been implemented in CLU on a single-user Vax (see Section 2.5). In addition to parsing specifications, this program checks various context sensitive constraints and provides mechanisms for "expanding" assumptions, importations, and inclusion. This checker is an interim tool. We designed our specification language in tandem with an editing and viewing tool. Many language design decisions were influenced by the presumption that specifications would be produced and read interactively using this tool. A first design of this tool is complete (see Section 2.4), but implementation has yet to begin. It will not begin until we have a high- resolution high-bandwidth display for our single-user Vaxes.

Construction of a primitive checker for the CLU interface language is scheduled to begin in June 1983. We plan to complete it by the end of the summer (see Section 2.5).

As part of the RR project, we are in the process of implementing a term rewriting system, REVE, that we hope will provide much of the theorem-proving capability needed for analyzing specifications. The definition of the Larch Shared Language calls for a number of checks for which there can be no effective procedure. We have what we believe are useful procedures, based on sufficient or necessary (but not both) conditions, for many of these checks, e.g., consistency. We are working on procedures for the others.

2.2. The Two-tiered Approach to Specification

The two-tiered approach to specifying programs separates the specification of underlying abstractions from the specification of state transformations. We use a "shared" specification language to describe underlying abstractions, and an "interface" specification language to describe state transformations. Furthermore, we intentionally make the interface specification language dependent on a target programming language. This decision allows us to keep separate the description of programming language independent issues from the description of programming language dependent ones, e.g., side effects, error handling, and resource allocation. For example, if we were to specify machine arithmetic, we would describe ideal arithmetic in the shared language, and we would describe boundary conditions constrained by word and memory size in an interface language.

The specification of a program module is written in an interface language of which the shared language is a subset. An interface specification contains two parts: a "shared language component" (bottom tier) and an "interface language component" (top tier). These two components correspond to the two tiers in our approach.

The invention and description of key abstractions should be done in the shared language. We expect most of the effort involved in writing a specification to be invested in the shared language component. The interface language component should deal only with programming language dependent issues. One reason for separating the two language components is that we expect many shared language components to be reusable by different interface language components. Some of them will be developed for particular applications; a few central ones will be useful in many applications.

2.3. The Larch Family of Specification Languages

Some important aspects of the Larch family of specification languages are:

Composability of specifications: We emphasize the incremental construction of specifications from other specifications. Larch has mechanisms for building upon and decomposing specifications as well as for combining specifications.

Emphasis on presentation: Reading specifications is an important activity. To assist in this process, we use composition mechanisms defined as operations on specifications, rather than on theories or models.

Interactive and integrated with tools: The Larch languages are designed for interactive use. They are intended to facilitate the interactive construction and incremental checking of specifications. The decision to rely heavily on support tools has influenced our language design in many ways.

Semantic checking: It is all too easy to write specifications with surprising implications. We would like many such specifications to be detectably ill-formed. Extensive checking while specifications are being constructed is an important aspect of our approach. Larch was designed to be used with a powerful theorem prover for semantic checking to supplement the syntactic checks commonly defined for specification languages.

Programming language dependencies factored: We feel that it is important to incorporate many programming-language-dependent features into our specification languages, but to isolate this aspect of specifications as much as possible. This prompted us to design a single shared language that could be combined in a uniform way with different interface languages.

Shared language based on equations: The shared language has a simple semantic basis taken from algebra. Because of the emphasis on composability, checkability and interaction, however, it differs substantially from the "algebraic" specification languages we have used in the past.

Interface languages based on predicate calculus: Each interface language is based on assertions written in first order predicate calculus with equality, and incorporates programming-language-specific features to deal with constructs such as side effects, exception handling, and iterators. Equality over terms is defined in the shared language; this provides the link between the two parts of a specification.

The crucial syntactic property of the shared language component is that it provides to the interface language component a set of "sort" identifiers and a set of "function" identifiers and function signatures. The function identifiers are composed to build "terms," which are used to write first-order "assertions." The sort identifiers and function signatures are used to "sort-check" terms much in the same way as type identifiers are used to type-check programs. The crucial semantic property of the shared language component is that it provides to the interface language component a theory of equality for terms.

A specification written in an interface language has three parts: a "header," a "body," and a "link" to the shared language component of the specification. The syntax of the header is based on the syntax of the programming language. For example, the types of the input and output arguments to a procedure are listed in the header information of a procedure specification as they would be in an implementation. The language of the assertions appearing in the body is based on the shared language component, plus special assertions, which are introduced to handle issues dependent on the semantics of the programming language. The meaning of the assertions is based on first-order predicate logic with equality, where equality is defined by the shared language component. The link identifies the shared language component to be used.

By explicitly including a shared language component in an interface specification, we gain the advantage that every symbol in an assertion is precisely defined within the specification language itself. In some other specification methods there is a reliance on an interpretation for symbols in an assertion, where the interpretation comes from outside the specification language. In contrast, some other methods [18][13] provide an assertion language defined within the specification language, but restrict the symbols to come from a fixed set of primitives. We gain the advantage that the user is able to provide just the symbols necessary to write the assertions in the body of a specification.

2.4. The Larch Editing Tool

Joe Zachary completed the design of a syntax-directed editing tool that facilitates the reading and writing of Larch specifications [21]. This work proceeded from the premise that the time has arrived to construct specialized tools to support the specification process.

Currently, a writer faces a series of sequential syntactic and semantic checks in producing a Larch specification. Context-free constraints are defined relative to free text; context-sensitive checks, relative to sentences in the context-free language; and semantic constraints, relative to syntactically correct text. This style of definition lends itself to a batch style of text production, with the editing and multiple checking phases distinct from one another.

This style of support is not satisfactory, because it focuses upon detecting errors some time after they are committed. Errors are often symptomatic of some more fundamental confusion and should be pointed out to the writer as they occur. For this reason, the editing and checking phases in the Larch editor are integrated. The editor is interactively involved in the incremental production and checking of specification text. A full range of error prevention, avoidance, and detection is available automatically at all times during the development of a specification.

Several central design decisions derive from this. The first concerns the extension of the Larch language. To allow full support at all times, partially completed specifications are included in the language. The definition of grammatical correctness is extended to include appropriate partial texts, and the context-sensitive and semantic constraints are redefined in terms of the extended language.

To ensure that the editor can always treat its text as a specification, the editor is syntax directed. By being syntax directed, it can restrict the form of developing specifications to be syntactically correct, ruling out an entire class of errors in the process. Text entry is simplified because the editor automatically supplies the skeleton forms of constructs. It also displays a formatted version of the specification at all times. The style of editing is derived from the Cornell Program Synthesizer [19].

The editor design also provides support for reading specifications. Larch specifications are written in small pieces that are combined to form the whole. Larch encourages this style of writing by providing a means of reference between traits. The constructs that provide references between traits specify textual transformations upon them, giving alternate ways of presenting the same text. The editor provides operations that interpret these constructs by constructing the different presentations. A specification is not regarded as static sequential text, but as a formal explanation designed for presentation by the editor.

Although the design of the editor is complete, it is as yet unimplemented. We expect that an implementation effort will commence within the next year.

2.5. Interim Checking Tools

Pistol is a syntax checker for the Larch shared language that was designed and implemented by Ron Kownacki during the past year. It provides interim machine support for the group's specification activities. In the style of the front end of a compiler, it parses specifications and subjects them to the various context-sensitive checks that are defined in the language manual. The checks are performed in layers, with the occurrence of context-free errors suppressing the performance of context-sensitive checking. Pistol also provides a primitive library facility and a specification pretty-printer.

The process of designing Pistol played an important role in the evolution of the shared language. The system was built using the group's software design methodology. During the design phase, Kownacki completed both a system specification to describe the intended behavior of the software and a structural specification to describe the implementation. The act of constructing these specifications resulted in several early indications of problems with the language. It also served to increase the group's awareness of the difficulties that arise solely as a result of the size of a specification.

The system has been of considerable value to SPD. Among other things, Pistol was used to debug a library of sample traits that was thought to be largely error-free. It was not. While the majority of the errors that were detected were simple oversights, some served to highlight subtle mistakes in the meaning of the specification.

Experimentation with the system has also allowed the group to identify certain problems with the shared language and its context-sensitive checks that probably could not have been discovered without machine support. One of Pistol's most attractive features is that it is very flexible in response to language changes. This quality has given the group a certain amount of freedom in the process of changing the shared language and the checking rules that are associated with it.

Now that the Larch shared language has been frozen, SPD is interested in expanding the functionality of Pistol in order to provide a more sophisticated interim tool to support its ongoing efforts at specifying. One natural extension would be to incorporate the semantic checks that must be applied to all specifications written in the shared language. We are currently studying the issues involved in checking for the properties of completeness and consistency.

It would also be useful to exploit other major benefits that are acquired through the use of formalism, such as the opportunity to provide facilities that aid the specifier in reasoning about the meaning of a specification. The incorporation of theorem proving machinery seems to be a logical step in this direction.

It is expected that these three semantic capabilities -- completeness checking, consistency checking and theorem proving -- will be provided within Pistol before the end of 1983. The implementation will rely heavily upon the technology of term-rewriting systems. REVE (see Section 3) will provide the necessary support for this project.

To support the interface tier of two-tiered specifications, a tool is being developed, by Kathy Yelick to support the CLU Interface Language. Current plans for the system include a syntax and static semantics checker. The language, based on the CLU Interface Language defined by Jeannette Wing, incorporates a number of syntactic changes believed to simplify the reading and writing of specifications. Included in this project is the writing of a user manual for the revised interface language and description of the tool. The work is expected to be complete in August 1983.

The interface language checker will be based on an LALR(1) parser produced using the YACC parser generator. Context sensitive checks will include type and sort checking of assertions. Terms in the interface language include object identifiers, quantified variables, and function identifiers from the trait language. Because the sort-correctness of a term in the interface language depends on its definition in the trait language, the syntax checker will use the sort checking facilities provided by Pistol to sort check most terms. In addition to the trait language terms, there are special assertions to handle such CLU concepts as objects, multiple termination, procedure arguments, and own variables. After completion of the checker, the language and tool will be informally evaluated by using them to write and check a relatively large specification.

3. THE REWRITE RULE LABORATORY

Term rewriting systems, also called rewrite rule systems, provide a model of computation that has the interesting and useful property of being directly applicable to obtaining decision procedures for equational theories [7]. Equational theories, in turn, supply the formal basis of our approach to specification.

Pierre Lescanne, a visiting researcher from the Centre de Recherche en Informatique de Nancy in France, developed a prototype system (in CLU) for generating and analyzing term rewriting systems. The system, REVE [14], includes a robust implementation of the Knuth-Bendix Completion Procedure [12] for "completing" a set of rewrite rules, two simplification orderings on terms for proving the uniform termination of such a rule set, and commands that rewrite terms for the purpose of proving theorems.

Using the Knuth-Bendix procedure, REVE can be used to prove both equational theorems and inductive theorems in the theory defined by a set of equations. The successful completion of Knuth-Bendix, together with uniform termination, provides a decision procedure for equational theorems. Huet and Hullot[8] extended the "inductionless induction" technique presented in [17] in proposing a modification to the Knuth-Bendix procedure that assists in the automatic proof of inductive theorems. REVE incorporates this modification to Knuth-Bendix, building on the formulation of the algorithm presented in [9]. The Knuth-Bendix implementation in REVE makes use of a unification algorithm with nearly linear-time complexity [15]. Uniform termination is required by Knuth-Bendix; REVE constructs the termination proof and runs Knuth-Bendix simultaneously. REVE differs from related systems (e.g., Affirm [16]) in that the proof of termination is nearly automatic.

Both of the simplification orderings used in REVE are extensions to a partial ordering on the function symbols, called a *precedence*, that appear in the rewriting system. One of them, the (lexicographical) recursive path ordering, is presented in [11] and is derived from the (standard) recursive path ordering [1]. This ordering assumes that the precedence is given *a priori*.

The other ordering on terms used in REVE is the (recursive) decomposition ordering [10]; REVE is the first system to use it. The two orderings yield similar results when comparing two terms. The recursive path ordering is more useful for termination proofs, since it can order an associativity equation while the decomposition ordering cannot. However, the decomposition ordering does not require an *a priori* precedence; rather, the decomposition ordering can assist in dynamically building the precedence as successive pairs of terms are compared. It is this feature, called *incrementality*, that allows REVE to prove termination semi-automatically. REVE uses the recursive path ordering for termination proofs, and the decomposition ordering to build the precedence.

Our group is in the early stages of using REVE. We have found the following paradigm useful: We construct an algebraic specification of an abstract data type, and attempt to complete the specification by invoking the Knuth-Bendix procedure. If Knuth-Bendix is successful in producing a decision procedure for the equational theory of the specification, we use REVE to prove theorems about the associated abstract type. The completed specification, together with any theorems, can then be used in conjunction with other completed specifications to produce decision procedures for the equational theories of more complex specifications involving many abstract types.

Randy Forgaard has almost completed a new production-quality implementation of REVE [2]. This version features improved modularity, faster algorithms, and increased functionality over the prototype program. The new implementation extends the Rewrite Rule Laboratory prototype presented in [3], in that all basic concepts and algorithms from rewrite rule theory that are used in REVE exist as separate procedural and data abstractions in the program. Future extensions to REVE, and implementation work by other researchers, can build on the extensive library of rewrite rule-related abstractions present in the Laboratory. This can lead to faster development of new algorithm implementations, facilitate the exchange of software between research groups, and encourage communication and cooperation between the researchers comprising the international rewrite rule community.

The user interface to the new REVE was developed by Ellen Frederiksen, an undergraduate. The interface is "friendlier" than that of the prototype REVE, and incorporates a history mechanism, allowing the user to "undo" an invocation of Knuth-Bendix. This is useful for removing pairs from the precedence that do not lead to a successful termination proof, and for retracting equations that are inconsistent with the rest of an equational theory.

During the past six months, our group has worked with a number of other research groups on REVE-related development. In France. Pierre Lescanne, Jean-Pierre Jouannaud, and their colleagues are: 1) developing an algorithm that checks for the well-definition of functions. as required by the inductionless induction method of [8]; 2) developing new methods of unification, serving to increase the set of algebraic specifications to which REVE can be applied; and 3) improving the simplification orderings used in REVE. At General Electric Corporate Research and Development, David Musser and Deepak Kapur have designed and implemented a LOGO-like language to serve as a command language for REVE and as a programming language for prototype implementations of new rewrite rule algorithms, have invented a new linear-time unification algorithm, and are exploring new inductionless-induction strategies. Mandayam K. Srivas, Jay Hsiang, and their students at the State Univ. of New York at Stony Brook, are implementing associative-commutative unification, and are looking at a decision procedure for the abstract type "boolean." All development efforts described above that are successful will be considered for inclusion in REVE; our group at MIT will continue to serve as the coordinator of future additions to REVE.

In August of this year, we will meet with the above research groups and other interested parties at a five-day mini-conference in New York. Half of the meeting will be a discussion of the rewrite rule research of the participants, and the other half will be devoted to a study of REVE's implementation and the integration of new extensions to REVE that are brought to the meeting.

In addition to the above colleagues, the prototype REVE is also being used by Nachum Dershowitz at the Univ. of Illinois and by researchers at the Univ. of Texas, Cornell Univ., Aerospace Corporation, Univ. of Colorado, and Univ. of Manchester in England. A number of others have expressed interest in obtaining the production-quality REVE when available. REVE is distributed free of charge to all who request it.

References

1. Dershowitz, N. "Orderings for Term-Rewriting Systems," Theoretical Computer Science, 17, North-Holland, 1982, 279-301.

2. Forgaard, R. "A Program for Generating and Analyzing Term Rewriting Systems," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected 1983.

3. Goree, J. "Using Abstractions to Implement the Knuth-Bendix Completion Procedure," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1981.

4. Guttag, J.V. Horning, J.J. and Wing, J.W. "Some Notes on Putting Formal Specifications to Productive Use," Science of Computer Programming, 2, (December 1982), 53-68.

5. Guttag, J.V. and Horning, J.J. "An Introduction to the Larch Shared Language," Proceedings IFIP Congress '83, Paris, France, 1983.

6. Guttag, J.V. and Horning, J.J. "Preliminary Report on the Larch Shared Language," Xerox Palo Alto Research Center Technical Report, Palo Alto, CA, (to appear 1983).

7. Huet, G. and Oppen, D.C. "Equations and Rewrite Rules: A Survey," Formal Languages: Perspectives and Open Problems, R. Book (ed., Academic Press, 1980.

8. Huet, G. and Hullot, J.-M. "Proofs by Induction in Equational Theories with Constructors," Proceedings 21st Symposium on Foundations of Computer Science, Los Angeles, CA, October 1980, 96-107.

9. Huet, G. "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm," J. Computer and System Sciences, 23, 1, (August 1981) 11-21.

10. Jouannaud, J.-P., Lescanne, P. and Reinig, F. "Recursive Decomposition Ordering," Proceedings of Formal Description of Programming Concepts II, IFIP TC-2 Working Conference, Garmisch-Partenkirchen, June 1982.

11. Kamin, S. and Levy, J.-J. "Attempts for Generalizing the Recursive Path Orderings," unpublished manuscript, February 1980.

12. Knuth, D.E. and Bendix, P.B. "Simple Word Problems in Universal Algebras," Computational Problems in Abstract Algebra, L. Leech, (ed.), Pergamon, Oxford, 1969, 263-297.

13. Jones, C.B. Software Development: A Rigorous Approach, Prentice-Hall, 1980.

14. Lescanne, P. "Computer Experiments with the REVE Term Rewriting System Generator," Proceedings 10th ACM Symp. on Principles of Programming Languages, Austin, TX, January 1983, 99-108.

15. Martelli, A. and Montanari, U. "An Efficient Unification Algorithm," ACM Transactions on Programming Languages and Systems, 4, 2, (April 1982), 258-282.

16. Musser, D.R. "Abstract Data Type Specification in the Affirm System," IEEE Transactions on Software Engineering, 6, 1, (January, 1980), 24-32.

17. Musser, D.R. "On Proving Inductive Properties of Abstract Data Types." Proceedings 7th ACM Symp. on Principles of Programming Languages, Las Vegas, NV, January 1980, 154-162.

18. Robinson, L. and Roubine, O. "SPECIAL--A Specification and Assertion Language," Stanford Research Institute, TR CSL-46, Stanford, CA, January 1977.

19. Teitelbaum, T. and Reps, T. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," CACM 24, 9, (September 1981), 563-573.

20. Wing, J.M. "A Two-Tiered Approach to Specifying Programs," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

21. Zachary, J. "A Syntax-Directed Tool for Constructing Specifications," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, March 1983.

Publications

1. Guttag, J.V., Horning, J.J. and Wing. J.M. "Some Notes on Putting Formal Specifications to Productive Use," Science of Computer Programming, 2 (December 1982), 53-68.

2. Guttag, J. V., Kapur, D. and Musser, D. R. "On Proving Uniform Termination and Restricted Termination of Rewriting Systems," SIAM Journal of Computing, 12, 1, (February 1983).

3. Guttag, J. V. Kapur, D. and Musser, D. R. "Derived Pairs, Overlap Closures, and Rewrite Dominoes: New Tools for Analyzing Term Rewriting Systems," Proceedings of the 1982 ICALP Conference, Lecture Notes in Computer Science, Springer-Verlag, July 1982.

4. Guttag, J.V. and Horning, J.J. "Preliminary Report on the Larch Shared Language," Xerox Palo Alto Research Centers Technical Report, Palo Alto, CA, (to appear 1983).

5. Lescanne, P. "Computer Experiments With the REVE Term Rewriting System Generator," Proceedings 10th ACM Symp. on Principles of Programming Languages, Austin, TX, January 1983, 99-108.

Theses Completed

1. Frederiksen, J. E. "User Interface of a System for Term Rewriting," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

2. Fuget, C. "A Test Suite for the C Compiler," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

3. Schaaf, R. "The Design of an Integrated and Incremental Environment for Software Development," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

4. Wing, J. "A Two-Tiered Approach to Specifying Programs," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, May 1983.

5. Zachary, J. "A Syntax Directed Tool for Constructing Specifications," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, March 1983.

Talks

1. Guttag, J. "The Larch Specification Language,"
   Phillips Research, Brussels, Belgium
   Danish Datamatics Center
   Xerox Palo Alto Research Centers.

2. Guttag, J. "The Prospects for Formal Specifications"
   State University of New York at Stony Brook
   University of Rhode Island.

3. Guttag, J. "Where and Whither Formal Specifications"
   Xerox Software Forum
   IBM Information Exchange Program.

4. Lescanne, P. "The REVE Term Rewriting System Generator," Bell Labs
   University of Illinois
   General Electric Corporate Research and Development
   1983 POPL Conference.

5. Wing, J. "A Two-Tiered Approach to Specifying Programs,"
   SUNY Stony Brook
   University of Toronto
   University of Maryland
   University of Southern California
   Carnegie Mellon University
   Harvard University.

THEORY OF DISTRIBUTED SYSTEMS

Academic Staff

N. A. Lynch, Group Leader

Graduate Students

B. Coan	J. Lundelius
J. Goree	E. Stark

Support Staff

E. Pothier

1. OVERVIEW

This year, the Theory of Distributed Systems Group worked on various problems in the theory of distributed computing, especially problems involving failures and problems involving time. Areas of progress were: (1) impossibility of distributed consensus (2) approximate agreement, (3) clock synchronization, (4) nested transactions and orphan detection, (5) models for distributed computing which incorporate time, (6) foundations of a theory of specification for distributed systems, and (7) Byzantine Generals.

2. IMPOSSIBILITY OF DISTRIBUTED CONSENSUS

A major accomplishment was the discovery of an interesting and fundamental impossibility result: the impossibility of reaching consensus among asynchronous distributed processes, in the presence of even a single, very simple fault. The asynchronous model is a very general, natural one to use for algorithm development. (In fact, there have been various attempts to design distributed consensus algorithms that would work within this model.) This result points out fundamental limitations on its use. The paper [9], based on this result, was presented as the lead-off paper at this year's Symposium on Principles of Database Systems.

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing. It is at the core of many of algorithms for distributed data processing, distributed file management, and fault-tolerant distributed applications.

A well-known form of the problem is the "transaction commit problem" which arises in distributed database systems [4], [11], [14], [15], [16], [17], [27], [28], [29], [30]. The problem is for all the data manager processes which have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or to discard them. The latter action might be necessary, for example, if some data managers were, for any reason, unable to carry out the required transaction processing. Whatever decision is made, all data managers must make the same decision in order to preserve the consistency of the database.

Reaching the type of agreement needed for the "commit" problem is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a number of possible faults such as process crashes, network partitioning, and lost, distorted or duplicated messages. One can even consider more Byzantine types of failure [2], [5], [6], [13], [20], [21], [24], in which faulty processes might go completely haywire, perhaps even sending messages according to some malevolent plan. One therefore wants an agreement protocol which is as reliable as possible in the presence of such faults. Of course, any protocol can be overwhelmed by faults that are too frequent or too severe, so the best that one can hope for is a protocol which is tolerant to a prescribed number of "expected" faults.

In this paper, we show the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death. We do not consider Byzantine failures, and we assume that the message system is reliable — it delivers all messages correctly and exactly once. Nevertheless, even with these assumptions, the stopping of a single process at an inopportune time can cause any distributed commit protocol to fail to reach agreement. Thus, this important problem has no robust solution without further assumptions about the computing environment or still greater restrictions on the kind of failures to be tolerated!

Crucial to our proof is that processing is completely asynchronous, that is, we make no assumptions about the relative speeds of processes nor about the delay time in delivering a message. We also assume that processes do not have access to synchronized clocks, so algorithms based on timeouts, for example, cannot be used. (In particular, the solutions in [4] are not applicable.) Finally, we do not postulate the ability to detect the death of a process, so it is impossible for one processes to tell whether another has died (stopped entirely) or is just running very slowly.

Our impossibility result applies to even a very weak form of the *consensus problem*. Assume every process starts with an initial value in $\{0, 1\}$. A nonfaulty process decides on a value in $\{0, 1\}$ by entering an appropriate decision state. All nonfaulty processes which decide are required to choose the same value. For the purpose of the impossibility proof, we require only that *some* process eventually make a decision. (Of course, any algorithm of interest would require that all nonfaulty processes make a decision.) The trivial solution in which, say, 0 is always chosen is ruled out by stipulating that both 0 and 1 are possible decision values, although perhaps for different initial configurations.

Our system model is rather strong so as to make our impossibility proof as widely applicable as possible. Processes are modeled as automata (with possibly infinitely many states) which communicate by means of messages. In one atomic step, a process can attempt to receive a message, perform local computation based on whether or not a message was delivered to it and if so on which one, and send an arbitrary but finite set of messages to other processes. In particular, an "atomic broadcast" capability is assumed, so a process can send the same message in one step to all other processes with the knowledge that if any nonfaulty process receives the message, then all the nonfaulty processes will. Every message is eventually delivered as long as the destination process makes infinitely many attempts to receive, but messages can be delayed arbitrarily long and delivered out of order.

The asynchronous commit protocols in current use all seem to have a "window of vulnerability" — an interval of time during the execution of the algorithm in which the delay or inaccessibility of a single process can cause the entire algorithm to wait indefinitely. It follows from our impossibility result that every commit protocol has such a "window", confirming a widely-believed tenet in the folklore.

This result has already generated several follow-up results by other researchers. Dolev, Dwork and Stockmeyer [3] have explored the sensitivity of our result to the assumptions we made on the model. Ben-Or [1], in an attempt to circumvent the limitation we discovered, has devised a clever probabilistic algorithm which solves the problem. Rabin [25] has improved the time efficiency of an algorithm similar to Ben-Or's, by utilizing certain cryptographic capabilities.

There are other possible approaches besides probabilistic algorithms, for avoiding the serious limitation described by this result. In certain situations, one could use approximate rather than exact agreement. Or, one could use algorithms that are not purely asynchronous, but rather use time in significant ways. We have been exploring both of these possibilities, and discuss them below.

3. APPROXIMATE AGREEMENT

In contrast to the impossibility result for simple consensus, new and simple algorithms have been obtained for approximate agreement on real values. A paper based on this work, [7], has been submitted to a conference.

In this work, we consider a variant on the distributed consensus problem, in which processes start with arbitrary real values rather than bit values or values from some bounded range. The object is for nonfaulty processes to agree on a value, in spite of the presence of a small number of faults. These can even be "Byzantine" types of faults - completely arbitrary, possibly malicious behavior. We assume a model in which processes can send messages containing arbitrary real values, and can store arbitrary real values as well.

We consider the case where approximate agreement suffices. Thus, we require algorithms in which all processes are guaranteed eventually to halt, and the following two conditions hold:

(a) Agreement

All nonfaulty processes eventually halt with values that are the same to within some small predetermined ϵ.

(b) Validity

The value output by any nonfaulty process must be in the range of initial values of the nonfaulty processes.

Thus, in particular, if all nonfaulty processes should happen to start with the same initial value, the final values are all required to be the same as that common initial value.

We consider both the synchronous and asynchronous versions of the problem.

First, we obtain a simple and rather efficient algorithm for the synchronous version. This algorithm works by successive approximation, with a provable convergence rate which depends on the ratio between the number of faults and the total number of processes. The algorithm is only guaranteed to converge in the case where the total number of processes is more than three times the number of possible faults. At every round of the algorithm, each process applies a kind of "averaging" function to the previous round's values of all processes. This averaging function operates by discarding a small fixed number of smallest and of largest values, before computing the mean (or midpoint). Termination is achieved using a simple binary Byzantine agreement on whether to halt. We also show that no such algorithm is possible in the case where the total number of processes is at most three times the number of possible faults.

Second, we extend the algorithm to the asynchronous case. In this case, the rate of convergence is slower, and the total number of processes required by our algorithm is more than five times the number of possible faults. The algorithm is based on the same averaging function as the synchronous algorithm. In the asynchronous case, we cannot use the simple trick of Byzantine agreement to decide on halting, since Byzantine agreement has been shown [9] to be impossible in the asynchronous case. Instead, we use another trick, which insures that all processes halt, but permits different processes to halt at different times.

An advantage of the simple convergence algorithm used here, over the usual Byzantine Agreement algorithms, is that it tolerates transient faults to a much greater extent.

We have also obtained lower bound results that show, under certain restrictions, that the rate of convergence of estimates yielded by our algorithm is optimal.

The problem of reaching approximate agreement on a real value appears to be a key component of the problem of synchronizing real-valued local clocks. Therefore, it appears that our approximate agreement algorithms have application to software clock synchronization in the presence of faults. These results are now being worked out in [22], and will be discussed further in the following section.

4. CLOCK SYNCHRONIZATION

Jennifer Lundelius is working on a Masters' thesis on clock synchronization. This research effort is directed toward exploring the problem of synchronizing clocks of processes in a distributed system. We have begun to formulate a series of progressively more accurate models. Within each model, we want to prove lower bounds on how closely processes' clocks can be synchronized (agreement condition), and present algorithms that achieve that bound. We are also concerned with making sure the new values of the clocks bear some relation to the previous values (validity condition).

We model the situation in which n processes communicate by sending messages. The message system is reliable and delivers all messages within a fixed amount of time plus or minus a bounded amount of uncertainty. Each process has a read-only clock, (a monotone increasing function from real numbers to real numbers), and a local correction variable which it can change in order to reset the clock. Processes are modeled as automata with states and transition functions. In one step, a process can receive messages, read its clock, change state and send out a finite number of messages. An execution of a system is represented as a sequence of events occurring at specific times, where an event is the delivery of a message or a process taking a step.

The dimensions along which the models differ include:

- introducing error in the clock rates - that is, allowing the clocks to be fast or slow,

- considering the time taken by each step of the processes. When this is nonzero, a buffer of waiting messages is required.

- allowing some number of the processes to be faulty, including the possibility of Byzantine failures. Presumably, there is some ratio of faulty to nonfaulty processes above which "synchronization" is not possible, as in the classical Byzantine Agreement problem.

So far, we have considered the fault-free case with no clock drift and instantaneous step time. The main result we have obtained is that it is impossible to synchronize n processes any closer than $(2 - 2/n)\epsilon$, where ϵ is the uncertainty in the message delay. The proof technique consists of assuming that there is an algorithm that can synchronize more closely, and considering various executions of the system of processes in which certain parameters are altered as much as possible within the uncertainty of the system, until finally a contradiction is reached.

There is a very simple algorithm which achieves exactly this degree of synchrony. Namely, each process sends a message containing its clock reading to each other process, estimates the other processes' clock values at a single time, assuming the average message delay, and sets its clock to the average of all the clock values, including its own. This algorithm is guaranteed to synchronize to within $(2 - 2/n)\varepsilon$; thus, the bound is tight. Moreover, the validity condition is satisfied.

In the case of Byzantine failures, the kind of averaging functions used in [7] are used in place of the ordinary mean. This yields clock synchronization algorithms which are tolerant to a small number of faults.

Current efforts are directed toward characterizing exactly what can be achieved in the Byzantine case. There is also still much work to be done in the fault-free case considering non-zero clock drift and non-zero process step time.

5. NESTED TRANSACTIONS AND ORPHAN DETECTION

Interesting examples of distributed algorithms designed to work in the presence of failures are the algorithm implementing nested transactions and the "orphan detection" algorithm, both used in the implementation of Argus. (See Programming Methodology Group for information about Argus.) N. Lynch [19] and J. Goree [10] worked on formal studies of the correctness of these algorithms.

In the past few years, there has been considerable research on concurrency control, including both systems design and theoretical study. The problem is roughly as follows. Data in a large (centralized or distributed) database is assumed to be accessible to users via transactions, each of which is a sequential program which can carry out many steps accessing individual data objects. It is important that the transactions appear to execute "atomically", i.e., without intervening steps of other transactions. However, it is also desirable to permit as much concurrent operation of different transactions as possible, for efficiency. Thus, it is not generally feasible to insist that transactions run completely serially. A notion of equivalence for executions is defined, where two executions are equivalent provided they "look the same" to all transactions and to all data objects. The serializable executions are just those which are equivalent to serial executions. One goal of concurrency control design is to insure that all executions of transactions be serializable.

Several characterization theorems have been proved for serializability; generally, they amount to the absence of cycles in some relation describing the dependencies among the steps of the transactions. A very large number of concurrency control algorithms have been devised. Typical algorithms are those based on two-phase locking [8], and those based on timestamps [12]. Although many of these algorithms are very different from each other, they can all be shown to be correct concurrency control algorithms. The correctness proofs depend on the absence-of-cycles characterizations for serializability.

More recently, it has been suggested [18], [23], [27] that some additional structure on transactions might be useful for programming distributed databases, and even for programming more general distributed systems. The suggested structure permits transactions to be nested. Thus, a transaction is not necessarily a sequential program, but rather can consist of (sequential or concurrent) sub-transactions. The intention is that the sub-transactions are to be serialized with respect to each other, but the order of serialization need not be completely specified by the writer of the transaction. This flexibility allows more concurrency in the implementation than would be possible with a single-level transaction structure consisting of sequential transactions. The general structure allows transactions to be nested to any depth, with only the leaves of the nesting tree actually performing accesses to data.

Transactions are often used not only as a unit of concurrency, but also as a unit of recovery. In a nested transaction structure, it is natural to try to localize the effects of failures within the closest possible level of nesting in the transaction nesting tree. One is naturally led to a style of programming which permits a transaction to create children, and to tolerate the reported failure of some of its children, using the information about the occurrence of the failures to decide on its further activity. The intention is that failed transactions are to have no effect on the data or on other transactions. This style of programming is a generalization of the "recovery block" style of [26] to the domain of concurrent programming. Indeed, this style seems to be especially suitable for programming distributed systems, since many types of failures of pieces of programs are likely to occur in such systems.

Reed's system uses multiple versions of data to implement nested transactions which tolerate failures of sub-transactions. Moss has abstracted away from Reed's specific implementation of nested transactions, and has presented a clear intuitive description of the nested transaction model. He has also developed an alternative implementation of the nested transaction model, based on two-phase locking. This model and implementation are fundamental to the Argus distributed computing language, now under development by Liskov's Group.

The basic correctness criteria for nested transactions seem to be clear enough, intuitively, to allow implementors a sufficient understanding of the requirements for their implementation. However, some subtle issues of correctness have arisen in connection with the behavior of failed sub-transactions. For example, the Argus group has decided that a pleasant property for an implementation to have is that all transactions, including even "orphans" (subtransactions of failed transactions), should see "consistent" views of the data (i.e., views that could occur during an execution in which they are not orphans). The implementation goes to considerable lengths to try to insure this property, but it is difficult for the implementors to be sure that they have succeeded.

It seems clear that some basic groundwork is needed before such properties can be proved. Namely, the theory already developed for concurrency control of single-level transaction systems without failures needs to be generalized to incorporate considerations of nesting and failures. The model needs to be formal, in order to allow careful specification of all the correctness requirements - the simple and intuitive ones, as well as the rather subtle ones.

The paper [19] begins to develop this groundwork. First, a simple "action tree" structure is defined, which describes the ancestor relationships among executing transactions and also describes the views which different transactions have of the data. A generalization of serializability to the domain of nested transactions with failures, is defined. A characterization is given for this generalization of serializability, in terms of absence of cycles in an appropriate dependency relation on transactions. A slightly simplified version of Moss' algorithm is presented in detail, and a careful correctness proof is given.

The style of correctness proof for the algorithm appears to be quite interesting in its own right. The description of the algorithm is presented in a series of levels, each of which is an "event-state" algebra with unary operations, and each (but the first) of which "simulates" the previous one. The basic problem statement is given as the highest level algebra, and successively lower levels provide increasing amounts of implementation detail. In particular, both the problem specification and the implementation are presented as the same kind of mathematical object, an event-state algebra. At every level, we want to present algorithms with the maximum possible amount of nondeterminism consistent with correctness, not forcing any unnecessary implementation decisions. Therefore, we do not describe algorithms in the usual way, using programs with specified flow of control. Rather, we present algorithms as collections of events with corresponding preconditions.

One novel aspect of the simulations we use, different from the usual notions of "abstraction" mappings, is that our simulations map single lower level states to sets of higher level states, rather than just single higher level states. (We call them "possibilities" mappings.) This extra flexibility seems quite convenient for many implementations, allowing the more "concrete" algebra sometimes to contain less information than the more "abstract" algebra. For example, it might be easy to prove correctness of an algorithm which maintains lots of auxiliary information. The correctness of an algorithm which maintains less information could be proved, in our model, by showing that it simulates the algorithm which maintains the auxiliary information.

While possibilities mappings are convenient for proving correctness of ordinary centralized algorithms, they produce their greatest payoff for distributed algorithms. Namely, a distributed algorithm is described as a special case of an event-state algebra, a "distributed algebra". In a distributed algebra, the state set is just a Cartesian product, with event preconditions and transitions defined componentwise. To show that a distributed algebra simulates some other "abstract" algebra, it suffices to define an appropriate possibilities mapping from the global states of the distributed algebra, to sets of states of the abstract algebra. It turns out to be extremely natural to describe such a mapping by first describing a possibilities mapping from the local state of each component to sets of abstract states. The image of a local state under this mapping just represents the set of possible global states consistent with the knowledge of the particular component. The possibilities for the entire distributed algebra are simply obtained by taking the intersection of the possibilities consistent with the knowledge of all the components.

It appears that this technique extends to give natural proofs of many algorithms, especially distributed algorithms, and thus warrants further investigation. Goree [10] presents a more complete (and slightly more general) development of the technique than is presented in this paper. Other related work is that of Stark [31]. He is carrying out a very general treatment of event-state algebras, incorporating considerations of modularity to a much greater extent than is present in this paper, and handling fairness and eventuality properties as well as safety properties.

John Goree's Masters' thesis was completed in January, and dealt with correctness of the "orphan detection" algorithm. This thesis defines a property called "view-serializability," which formalizes internal consistency for a system of nested atomic transactions. Internal consistency is a stronger condition than the usual notion of database consistency, because it takes into account the views of transactions which will never commit. In a distributed system, local aborts of remote subactions and crashes of nodes can generate *orphans:* active actions which are descendants of actions that have aborted or are guaranteed to abort. Because it is not always feasible or efficient to eliminate orphans immediately, special care is needed to insure that they see consistent system states when they are allowed to continue running. We investigate a particular dynamic detection strategy designed to detect orphans before they violate internal consistency. This algorithm piggybacks abort and crash information on the normal messages between nodes. We consider a simpler algorithm that only handles orphans arising from explicit aborts. We describe the simplified orphan detection algorithm at various levels of abstraction, using an algebraic model convenient for describing asynchronous systems. The highest-level model is specified in terms of a (virtual) global state. At this level of abstraction we require that the states generated by the model satisfy view-serializability. Lower-level models progressively localize the description of the algorithm's operation, and the lowest level of abstraction presents a fully distributed model of the (simplified) orphan detection scheme.

6. MODELS FOR DISTRIBUTED COMPUTING WHICH INCORPORATE TIME

If computing is done using a model that is not complete asynchronous, but rather makes some assumptions about relative rates of processes, or makes use of local clocks and assumptions about event times, then the limitations described in Section 2, above, do not apply. In fact, there are many consensus protocols in the literature which depend on assumptions involving time.

The correctness of some of these protocols seems less than obvious; it appears that there is a need for formal models for distributed systems that use time, of the same simplicity and rigor as now exist for synchronous and asynchronous systems. This need became quite evident when we began writing up proofs of some of the clock synchronization results; there simply did not seem to exist appropriate formal models for proving our results.

We have therefore begun development and study of formal models for distributed computing which use time explicitly. So far, a preliminary set of basic definitions for a simple automata-theoretic model has been devised. We hope that some version of this model will be usable in careful reasoning about the correctness of algorithms which use time.

7. FOUNDATIONS FOR A THEORY OF SPECIFICATION FOR DISTRIBUTED SYSTEMS

Gene Stark has been working on a Ph.D. dissertation entitled: "Foundations of a Theory of Specification for Distributed Systems." This work has two major foci: one, the development of a mathematical model of distributed/concurrent computer systems which incorporates as primitive notions the concepts of *hierarchy* and *modularity*, and which is independent of any particular programming or specification language; and two, the use of this model as the basis for investigating a particular approach to specifying the behavior of distributed systems, called *state-transition specification*. Hierarchy is concerned with viewing a single system at a number of levels of abstraction, whereas modularity is concerned with the formation of composite systems from a collection of independent components. Novel features of this work include: (1) the integration of the notions of hierarchy and especially modularity into a theory of specification; (2) development of a single mathematical model which provides abstract, language-independent definitions of the notions of implementation and correctness, and which can serve as a foundation upon which disparate specification techniques can be based and compared; (3) the ability to specify both safety (invariance) and liveness (eventuality) properties of systems, and to reason about these properties in a single framework; (4) the development, as an integral part of the technique of state-transition specification, of a systematic method for constructing proofs of correctness for implementations involving modules specified by this technique.

The fundamental notion that links the mathematical model to the real world is that of an *event*, which is an instantaneous observable occurrence during the operation of a computer system. Associated with each particular level of abstraction is a partitioning of the events of a system into a collection of *event classes*. Examples of event classes are: the class of all events in which the variable x gets set to three, and the class of all events in which process A submits a message to a transmission system for delivery to process B. The collection of all event classes for a system at a particular level of abstraction is called the *interface* of the system. Associated with an interface is a collection of possible *observations*, each of which associates an event class with each instant of time, where each time instant corresponds to a point on the real line. Thus each observation is the record of the behavior of a system during a particular execution, and the interface of a system is the "syntax" from which observations are formed. The set of all possible observations that can be produced by a particular system is called the *behavior* of that system.

Hierarchy and modularity are captured by the formalism in the form of *abstraction* and *decomposition maps*, both of which are a kind of projections on observations. An abstraction map maps an observation expressed in terms of a fine-grained classification of the system events into a corresponding observation expressed in terms of a coarser-grained classification; thus modeling the passage from a level of more detail to one of less. A decomposition map is a vector of projections, each of which takes an observation for a "composite" system and produces a version of that observation which is "localized" to a single module in the system, in much the same way that a local view of the operation of a complex electronic circuit by placing the probes of a logic analyzer at the pins of a single integrated circuit package. The notions of implementation and correctness are defined in terms of abstraction and decomposition maps. The problem of system specification becomes the problem of defining which are the "good" observations for a system.

The interface of a system can be viewed as an alphabet, and each observation for that interface as a (possibly infinite) string over that alphabet. The set of "good" observations for a system can therefore by viewed as a language over the system interface. A classical method of defining a language is by an automaton that generates it. The technique of state-transition specification is based on this analogy. The good observations for a system are described by introducing a set of conceptual states of the system, and defining a kind of automaton with this state set, each computation of which generates an observation. The set of conceptual states and associated automaton are introduced merely as a tool for describing the observable aspects of system behavior; they need not have anything to do with the actual implementation of the system. The description of the automaton constitutes the specification of the safety properties of the system. Liveness properties are expressed by an additional predicate on computation, which defines certain computations of the automaton to be "valid" and the remainder to be "invalid." An observation is defined to be "good" if and only if it can be generated by some valid computation of the automaton.

The utility of the state-transition approach to specification is tested through application to example problems, including a distributed resource management system and a reliable message transmission system which uses the alternating bit protocol. In each example, specifications are given for an abstract module to be implemented, and for component modules which are used to implement the abstract module. These specifications are used in rigorous proofs that the implementations are correct. Examination of these proofs yields some interesting proof-structuring principles that should be generally useful.

8. BYZANTINE AGREEMENT

Brian Coan has been investigating the problem of reaching Byzantine agreement on long values in a synchronous system. This is the problem of reaching agreement in a system of processes some of which may fail. A possibly faulty commander distributes a value to each process. After executing the Byzantine agreement algorithm, each process must choose an answer. If the commander distributed its value correctly, all correct processes must agree on that value. In any event, all correct processes must agree on some common value. The processes communicate over a network that is fully connected, reliable, and identifies the sender of each message. Process failures can produce arbitrary, even malicious, behavior. This problem was first defined by Pease, Shostack, and Lamport [24] who show that algorithms exist only when the total number of processes exceed the number of faulty processors by more than a factor of three. A result of Lynch and Fischer [20] shows that in any algorithm the number of rounds of communication must exceed the number of faulty processes.

A basic building block in Byzantine agreement algorithms is the binary Byzantine agreement algorithm -- agreement on one bit. An algorithm due to Turpin [32] reaches Byzantine agreement on an arbitrary message value using the binary Byzantine agreement algorithm and a small amount of additional overhead (two rounds of message exchange). This algorithm is important because it is the best known with respect to the number of message bits required to reach Byzantine agreement on a long value. Coan has developed an improved version of this algorithm that removes a restriction that Turpin imposes on the behavior of the binary Byzantine algorithm used. Turpin's restriction increases the constraints on the answer of the correct processes when the commander is faulty. Therefore, Turpin's construction may be incompatible with some binary Byzantine generals algorithms.

An important feature of both of these algorithms is that they solve a complex distributed problem by decomposing it into simpler subproblems. In the past, the technique of decomposing a problem has been used with more success in sequential algorithms than in distributed algorithms. The decompositions here are evidence that the binary Byzantine generals algorithm may prove to be a important building block in many distributed algorithms.

We conducted a research seminar in the Spring of 1983 entitled "Coping with Failure in Distributed Algorithms". This seminar surveyed various techniques for introducing fault-tolerance into distributed algorithms. In particular, we went through most of the significant work that has been done on the Byzantine Agreement problem. Several students in the seminar carried out independent research on aspects of Byzantine agreement.

9. FUTURE PLANS

The Theory of Distributed Systems Group plans to continue working on issues involving reliability and timing in distributed systems.

(a) Computing Using Time:

We will complete development of formal models for computing using time, and attempt to prove fundamental results about the power of models that use time in various ways. We will use these models to prove correctness of some distributed algorithms that use timeouts. (Particular examples are the timeout-based consensus algorithms of Dolev and Strong [4] and of Rabin [25]. Other examples arise in network fault-detection algorithms.) We will consider the usability of these models as semantic models for real-time programming languages.

We will continue studying the basic capabilities of software clock synchronization.

(b) Reliability in Distributed Algorithms

We plan to look further at the orphan algorithm, to try to better understand its behavior in the presence of node crashes. More generally, we will seek a better understanding of techniques for achieving data consistency in the face of failures that lose information. We will also consider techniques for achieving various kinds of reliability in networks.

(c) Verification of Distributed Programs

We will try to determine if the methodology we have used for the multi-level proof of the nested transaction and orphan algorithms is more generally useful for distributed program verification.

(d) High-Level Constructs for Efficient and Reliable Distributed Computing

We would like to understand an appropriate framework for distributed transaction processing in the presence of network partitioning (perhaps even as the normal mode of operation). We would also like to understand the high-level primitives that are necessary to describe the strict timing and reliability requirements for applications such as avionics.

Some specific plans for this coming year include a postdoctoral visit by Dr. Cynthia Dwork, a new course in distributed algorithms, and possibly another research seminar.

References

1. Ben-Or, M. "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," Proceedings of Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 17-19, 1983.

2. DeMillo, R. A., Lynch, N. A. and Merritt, M. J. "Cryptographic Protocols," Proceedings 14th ACM Symposium on Theory of Computing, 1982, 383-400.

3. Dolev, D., Dwork, C. and Stockmeyer, S. "On the Minimal Synchronism Needed for Distributed Consensus," (Extended Abstract).

4. Dolev, D. and Strong, R. "Distributed Commit With Bounded Waiting," Proceedings 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems, July 1982.

5. Dolev, D. and Strong, R. "Polynomial Algorithms for Byzantine Agreement," Proceedings 14th ACM Symposium on Theory of Computing, 1982, 401-407.

6. Dolev, D., Fischer, M., Fowler, R., Lynch, N. and Strong, R. "Efficient Byzantine Agreement Without Authentication," to appear in Information and Control.

7. Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W. "On Reaching Approximate Agreement in the Presence of Faults," submitted for publication.

8. Eswaren, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. "The Notions of Consistency and Predicate Locks in a Database System," Communications of the ACM 19(11) (November 1976), 624-633.

9. Fischer, M., Lynch, N. and Paterson, M. "Impossibility of Distributed Consensus with One Faulty Process," Proceedings of Second ACM Symposium on Principles of Database Systems, 1983.

10. Goree, J. "Internal Consistency of A Distributed Transactions System with Orphan Detection," MIT/LCS/TR-286, MIT Laboratory for Computer Science, Cambridge, MA, January 1983.

11. Garcia-Molina, H. "Elections in a Distributed Computing System," IEEE Transactions on Computers C-31(1), (1982).

12. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM 21(7), (July 1978), 558-565.

13. Lamport, L., Shostak, R. and Pease, M. "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, 4(3) (1982), 382-401.

14. Lampson, B. "Replicated Commit," CSL Notebook Entry, Xerox Palo Alto Research Center, Palo Alto, CA, 1981.

15. Lampson, B. and Sturgis, H. "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center Manuscript, Palo Alto, CA, 1979.

16. LeLann, G. "Private Communication," to appear.

17. Lindsay, B. et al. "Notes on Distributed Databases," IBM Research Report RJ2571, 1979.

18. Liskov, B. and Scheifler, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," 1982 Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 1982, Albuquerque, NM, 7-19.

19. Lynch, N.A. "Concurrency Control for Resilient Nested Transactions," Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1982

20. Lynch, N. A. and Fischer, M. "A Lower Bound For The Time To Assure Interactive Consistency," Information Processing Letters 14(4), (June 1982), 183-186.

21. Lynch, N. A., Fischer, M. J. and Fowler, R. J. "A Simple and Efficient Byzantine Generals Algorithm," Proceedings Second Symposium on Reliability in Distributed Software and Database Systems, July 1982, 46-52.

22. Lynch, N. A. and Lundelius, J. "On Clock Synchronization," to appear.

23. Moss, J.E.B. "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, April 1981. Also MIT/LCS/TR-260, MIT Laboratory for Computer Science.

24. Pease, M., Shostak, R. and Lamport, L. "Reaching Agreement in the Presence of Faults," Journal of the Association for Computing Machinery, 27(2), (April 1980), 228-234.

25. Rabin, M. "Randomized Byzantine Generals," to appear.

26. Randell, B. "System Structures for Software Fault Tolerance," Proceedings International Conference on Reliable Software, SIGPLAN Notices, 10, 6, (April 1975), 437-457. Also in IEEE Transactions on Software Engineering, 1,2 (June 1975), 220-232.

27. Reed, D. P. "Naming and Synchronization in a Decentralized Computer System," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1978.

28. Rosenkrantz, D., Stearns, R. and Lewis, P. "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems 3,(2), (June 1978), 178-198.

29. Skeen, D. "A Decentralized Termination Protocol," Proceedings 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems, July 1981, Pittsburgh, PA, 27-32.

30. Skeen, D. and Stonebraker, M. "A Formal Model of Crash Recovery in a Distributed Database Systems," Proceedings of 5th Berkeley Workshop on Distributed Data Management and Computer Networks, 1981, 129-142.

31. Stark, E. "Foundations of a Theory of Specification for Distributed Systems," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, to appear.

32. Turpin, R. "Reducing Byzantine Agreement on Messages From An Arbitrary Domain To Binary Byzantine Agreement," to appear.

Publications

1. Lynch, N. "Multilevel Atomicity," Proceedings of the ACM Symposium on Principles of Database Systems, Marina del Rey Hotel, Los Angeles, CA, March 1982, 63-69.

2. Lynch, N. "Concurrency Control for Resilient Nested Transactions," Proceedings of Second ACM Symposium on Principles of Database Systems, March 1983.

3. Fischer, M., Lynch, N. and Paterson, M. "Impossibility of Distributed Consensus with One Faulty Process," Proceedings of Second ACM Symposium on Principles of Database Systems, 1983.

4. Dolev, D., Lynch, N., Pinter, S. , Stark, E. and Weihl, W. "On Reaching Approximate Agreement in the Presence of Faults," to appear.

Thesis Completed

1. Goree, J. "Internal Consistency Of A Distributed Transaction System With Orphan Detection," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983

Theses in Progress

1. Stark, E. "Foundations of a Theory of Specification for Distributed Systems," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1983.

2. Lundelius, J. "Synchronizing Clocks in A Distributed System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1985.

Talks

1. Lynch, N. A. "Probabilistic Analysis of a Network Resource Allocation Algorithm," AMS Workshop on Probabilistic Algorithms, Summer 1982.

2. Lynch, N. A. "Impossibility of Distributed Consensus,"
   Harvard University, Fall 1982
   Cornell University, Fall 1982.

3. Allchin, J. "Concurrency Control for Resilient Nested Transactions," 2nd Annual Symposium on Principles of Database Systems, Atlanta, GA, March 21-22, 1983.

4. Lynch, N. A. "On Reaching Distributed Agreement"
   General Instruments, April,1983
   Apollo Computer, June 1983
   General Telephone and Electronics, June 1983.

PRECEDING PAGE BLANK-NOT FILMED

PUBLICATIONS

Technical Memoranda

TM-10[9]	Jackson, James N. Interactive Design Coordination for the Building Industry, June 1970, AD 708-400
TM-11	Ward, Philip W. Description and Flow Chart of the PDP-7/9 Communications Package, July 1970, AD 711-379
TM-12	Graham, Robert M. File Management and Related Topics June 12, 1970, September 1970, AD 712-068
TM-13	Graham, Robert M. Use of High Level Languages for Systems Programming, September 1970, AD 711-965
TM-14	Vogt, Carla M. *Suspension of Processes in a Multi-processing Computer* System, September 1970, AD 713-989
TM-15	Zilles, Stephen N. An Expansion of the Data Structuring Capabilities of PAL, *October 1970, AD 720-761*
TM-16	Bruere-Dawson, Gerard Pseudo-Random Sequences, October 1970, AD 713-852
TM-17	Goodman, Leonard I. Complexity Measures for Programming Languages, September 1971, AD 729-011
TM-18	Reprinted as TR-85
TM-19	Fenichel, Robert R. A New List-Tracing Algorithm, October 1970, AD 714-522

[9] TMs 1-9 were never issued.

PUBLICATIONS

TM-20	Jones, Thomas L. A Computer Model of Simple Forms of Learning, January 1971, AD 720-337
TM-21	Goldstein, Robert C. The Substantive Use of Computers For Intellectual Activities, April 1971, AD 721-618
TM-22	Wells, Douglas M. Transmission Of Information Between A Man-Machine Decision System And Its Environment, April 1971, AD 722-837
TM-23	Strnad, Alois J. The Relational Approach to the Management of Data Bases, April 1971, AD 721-619
TM-24	Goldstein, Robert C. and Alois J. Strnad The MacAIMS Data Management System, April 1971, AD 721-620
TM-25	Goldstein, Robert C. Helping People Think, April 1971, AD 721-998
TM-26	Iazeolla, Giuseppe G. Modeling and Decomposition of Information Systems for Performance Evaluation, June 1971, AD 733-965
TM-27	Bagchi, Amitava Economy of Descriptions and Minimal Indices, January 1972, AD 736-960
TM-28	Wong, Richard Construction Heuristics for Geometry and a Vector Algebra Representation of Geometry, June 1972, AD 743-487
TM-29	Hossley, Robert and Charles Rackoff The Emptiness Problem for Automata on Infinite Trees, Spring 1972, AD 747-250
TM-30	McCray, William A. SIM360: A S/360 Simulator, October 1972, AD 749-365
TM-31	Bonneau, Richard J. A Class of Finite Computation Structures Supporting the Fast Fourier Transform, March 1973, AD 757-787

TM-32	Moll, Robert An Operator Embedding Theorem for Complexity Classes of Recursive Functions, May 1973, AD 759-999
TM-33	Ferrante, Jeanne and Charles Rackoff A Decision Procedure for the First Order Theory of Real Addition with Order, May 1973, AD 760-000
TM-34	Bonneau, Richard J. Polynomial Exponentiation: The Fast Fourier Transform Revisited, June 1973, PB 221-742
TM-35	Bonneau, Richard J. An Interactive Implementation of the Todd-Coxeter Algorithm, December 1973, AD 770-565
TM-36	Geiger, Steven P. A User's Guide to the Macro Control Language, December 1973, AD 771-435
TM-37	Schonhage, A. Real-Time Simulation of Multidimensional Turing Machines by Storage Modification Machines, December 1973, PB 226-103/AS
TM-38	Meyer, Albert R. Weak Monadic Second Order Theory of Succesor is not Elementary-Recursive, December 1973, PB 226-514/AS
TM-39	Meyer, Albert R. Discrete Computation: Theory and Open Problems, January 1974, PB 226-836/AS
TM-40	Paterson, Michael S., Michael J. Fischer and Albert R. Meyer An Improved Overlap Argument for On-Line Multiplication, January 1974, AD 773-137
TM-41	Fischer, Michael J. and Michael S. Paterson String-Matching and Other Products, January 1974, AD 773-138
TM-42	Rackoff, Charles On the Complexity of the Theories of Weak Direct Products, January 1974, PB 228-459/AS

TM-43	Fischer, Michael J. and Michael O. Rabin Super-Exponential Complexity of Presburger Arithmetic, February 1974, AD 775-004
TM-44	Pless, Vera Symmetry Codes and their Invariant Subcodes, May 1974, AD 780-243
TM-45	Fischer, Michael J. and Larry J. Stockmeyer Fast On-Line Integer Multiplication, May 1974, AD 779-889
TM-46	Kedem, Zvi M. Combining Dimensionality and Rate of Growth Arguments for Establishing Lower Bounds on the Number of Multiplications, June 1974, PB 232-969/AS
TM-47	Pless, Vera Mathematical Foundations of Flip-Flops, June 1974, AD 780-901
TM-48	Kedem, Zvi M. The Reduction Method for Establishing Lower Bounds on the Number of Additions, June 1974, PB 233-538/AS
TM-49	Pless, Vera Complete Classification of (24,12) and (22,11) Self-Dual Codes, June 1974, AD 781-335
TM-50	Benedict, G. Gordon An Enciphering Module for Multics, S.B. Thesis, EE Dept., July 1974, AD 782-658
TM-51	Aiello, Jack M. An Investigation of Current Language Support for the Data Requirements of Structured Programming, S.M. & E.E. Thesis, EE Dept., September 1974, PB 236-815/AS
TM-52	Lind, John C. Computing in Logarithmic Space, September 1974, PB 236-167/AS
TM-53	Bengelloun, Safwan A. MDC-Programmer: A Muddle-to Datalanguage Translator for Information Retrieval, S.B. Thesis, EE Dept., October 1974, AD 786-754

TM-54	Meyer, Albert R. The Inherent Computation Complexity of Theories of Ordered Sets: A Brief Survey, October 1974, PB 237-200/AS
TM-55	Hsieh, Wen N., Larry H. Harper and John E. Savage A Class of Boolean Functions with Linear Combinatorial Complexity, October 1974, PB 237-206/AS
TM-56	Gorry, G. Anthony Research on Expert Systems, December 1974
TM-57	Levin, Michael On Bateson's Logical Levels of Learning, February 1975
TM-58	Qualitz, Joseph E. Decidability of Equivalence for a Class of Data Flow Schemas, March 1975, PB 237-033/AS
TM-59	Hack, Michael Decision Problems for Petri Nets and Vector Addition Systems, March 1975 PB 231-916/AS
TM-60	Weiss, Randell B. CAMAC: Group Manipulation System, March 1975, PB 240-495/AS
TM-61	Dennis, Jack B. First Version of a Data Flow Procedure Language, May 1975
TM-62	Patil, Suhas S. An Asynchronous Logic Array, May 1975
TM-63	Pless, Vera Encryption Schemes for Computer Confidentiality, May 1975, AD A010-217
TM-64	Weiss, Randell B. Finding Isomorph Classes for Combinatorial Structures, S.M. Thesis, EE Dept., June 1975
TM-65	Fischer, Michael J. The Complexity Negation-Limited Networks - A Brief Survey, June 1975

TM-66	Leung, Clement Formal Properties of Well-Formed Data Flow Schemas, S.B., S.M. & E.E. Thesis, EE Dept., June 1975
TM-67	Cardoza, Edward E. Computational Complexity of the Word Problem for Commutative Semigroups, S.M. Thesis, EE & CS Dept., October 1975
TM-68	Weng, Kung-Song Stream-Oriented Computation in Recursive Data Flow Schemas, S.M. Thesis, EE & CS Dept., October 1975
TM-69	Bayer, Paul J. Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees, S.M. Thesis, EE & CS Dept., November 1975
TM-70	Ruth, Gregory R. Automatic Design of Data Processing Systems, February 1976, AD A023-451
TM-71	Rivest, Ronald On the Worst-Case of Behavior of String-Searching Algorithms, April 1976
TM-72	Ruth, Gregory R. Protosystem I: An Automatic Programming System Prototype, July 1976, AD A026-912
TM-73	Rivest, Ronald Optimal Arrangement of Keys in a Hash Table, July 1976
TM-74	Malvania, Nikhil The Design of a Modular Laboratory for Control Robotics, S.M. Thesis, EE & CS Dept., September 1976, AD A030-418
TM-75	Yao, Andrew C. and Ronald I. Rivest K + 1 Heads are Better than K, September 1976, AD A030-008
TM-76	Bloniarz, Peter A., Michael J. Fischer and Albert R. Meyer A Note on the Average Time to Compute Transitive Closures, September 1976

TM-77	Mok, Aloysius K. Task Scheduling in the Control Robotics Environment, S.M. Thesis, EE & CS Dept., September 1976, AD A030-402
TM-78	Benjamin, Arthur J. Improving Information Storage Reliability Using a Data Network, S.M. Thesis, EE & CS Dept., October 1976, AD A033-394
TM-79	Brown, Gretchen P. A System to Process Dialogue: A Progress Report, October 1976, AD A033-276
TM-80	Even, Shimon The Max Flow Algorithm of Dinic and Karzanov: An Exposition, December 1976
TM-81	Gifford, David K. Hardware Estimation of a Process' Primary Memory Requirements, S.B. Thesis, EE & CS Dept., January 1977
TM-82	Rivest, Ronald L., Adi Shamir and Len Adleman A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, April 1977, AD A039-036
TM-83	Baratz, Alan E. Construction and Analysis of Network Flow Problem which Forces Karzanov Algorithm to $O(n^3)$ Running Time, April 1977
TM-84	Rivest, Ronald L. and Vaughan R. Pratt The Mutual Exclusion Problem for Unreliable Processes, April 1977
TM-85	Shamir, Adi Finding Minimum Cutsets in Reducible Graphs, June 1977, AD A040-698
TM-86	Szolovits, Peter, Lowell B. Hawkinson and William A. Martin An Overview of OWL, A Language for Knowledge Representation, June l977, AD A041-372
TM-87	Clark, David., editor Ancillary Reports: Kernel Design Project, June 1977

TM-88	Lloyd, Errol L. On Triangulations of a Set of Points in the Plane, S.M. Thesis, EE & CS Dept., July 1977
TM-89	Rodriguez, Humberto Jr. Measuring User Characteristics on the Multics System, S.B. Thesis, EE & CS Dept., August 1977
TM-90	d'Oliveira, Cecilia R. An Analysis of Computer Decentralization, S.B. Thesis, EE & CS Dept., October 1977, AD A045-526
TM-91	Shamir, Adi Factoring Numbers in O(log n) Arithmetic Steps, November 1977, AD A047-709
TM-92	Misunas, David P. Report on the Workshop on Data Flow Computer and Program Organization, November 1977
TM-93	Amikura, Katsuhiko A Logic Design for the Cell Block of a Data-Flow Processor, S.M. Thesis, EE & CS Dept., December 1977
TM-94	Berez, Joel M. A Dynamic Debugging System for MDL, S.B. Thesis, EE & CS Dept., January 1978, AD A050-191
TM-95	Harel, David Characterizing Second Order Logic with First Order Quantifiers, February 1978
TM-96	Harel, David, Amir Pnueli and Jonathan Stavi A Complete Axiomatic System for Proving Deductions about Recursive Programs, February 1978
TM-97	Harel, David, Albert R. Meyer and Vaughan R. Pratt Computability and Completeness in Logics of Programs, February 1978

TM-98	Harel, David and Vaughan R. Pratt Nondeterminism in Logics of Programs, February 1978
TM-99	LaPaugh, Andrea S. The Subgraph Homeomorphism Problem, S.M. Thesis, EE & CS Dept., February 1978
TM-100	Misunas, David P. A Computer Architecture for Data-Flow Computation, S.M. Thesis, EE & CS Dept., March 1978, AD A052-538
TM-101	Martin, William A. Descriptions and the Specialization of Concepts, March 1978, AD A052-773
TM-102	Abelson, Harold Lower Bounds on Information Transfer in Distributed Computations, April 1978
TM-103	Harel, David Arithmetical Completeness in Logics of Programs, April 1978
TM-104	Jaffe, Jeffrey The Use of Queues in the Parallel Data Flow Evaluation of "If-Then-While" Programs, May 1978
TM-105	Masek, William J. and Michael S. Paterson A Faster Algorithm Computing String Edit Distances, May 1978
TM-106	Parikh, Rohit A Completeness Result for a Propositional Dynamic Logic, July 1978
TM-107	Shamir, Adi A Fast Signature Scheme, July 1978, AD A057-152
TM-108	Baratz, Alan E. An Analysis of the Solovay and Strassen Test for Primality, July 1978
TM-109	Parikh, Rohit Effectiveness, July 1978

TM-110	Jaffe, Jeffrey M. An Analysis of Preemptive Multiprocessor Job Scheduling, September l978
TM-111	Jaffe, Jeffrey M. Bounds on the Scheduling of Typed Task Systems, September l978
TM-112	Parikh, Rohit A Decidability Result for a Second Order Process Logic, September 1978
TM-113	Pratt, Vaughan R. A Near-optimal Method for Reasoning about Action, September 1978
TM-114	Dennis, Jack B., Samuel H. Fuller, William B. Ackerman, Richard J. Swan and Kung-Song Weng Research Directions in Computer Architecture, September 1978, AD A061-222
TM-115	Bryant, Randal E. and Jack B. Dennis Concurrent Programming, October 1978, AD A061-180
TM-116	Pratt, Vaughan R. Applications of Modal Logic to Programming, December 1978
TM-117	Pratt, Vaughan R. Six Lectures on Dynamic Logic, December 1978
TM-118	Borkin, Sheldon A. Data Model Equivalence, December 1978, AD A062-753
TM-119	Shamir, Adi and Richard E. Zippel On the Security of the Merkle-Hellman Cryptographic Scheme, December 1978, AD A063-104
TM-120	Brock, Jarvis D. Operational Semantics of a Data Flow Language, S.M. Thesis, EE & CS Dept., December 1978, AD A062-997

TM-121	Jaffe, Jeffrey The Equivalence of R. E. Programs and Data Flow Schemes, January 1979
TM-122	Jaffe, Jeffrey Efficient Scheduling of Tasks Without Full Use of Processor Resources, January 1979
TM-123	Perry, Harold M. An Improved Proof of the Rabin-Hartmanis-Stearns Conjecture, S.M. & E.E. Thesis, EE & CS Dept., January 1979
TM-124	Toffoli, Tommaso Bicontinuous Extensions of Invertible Combinatorial Functions, January 1979, AD A063-886
TM-125	Shamir, Adi, Ronald L. Rivest and Leonard M. Adleman Mental Poker, February 1979, AD A066-331
TM-126	Meyer, Albert R. and Michael S. Paterson With What Frequency Are Apparently Intractable Problems Difficult?, February 1979
TM-127	Strazdas, Richard J. A Network Traffic Generator for Decnet, S.B. & S.M. Thesis, EE & CS Dept., March 1979
TM-128	Loui, Michael C. Minimum Register Allocation is Complete in Polynomial Space, March 1979
TM-129	Shamir, Adi On the Cryptocomplexity of Knapsack Systems, April 1979, AD A067-972
TM-130	Greif, Irene and Albert R. Meyer Specifying the Semantics of While-Programs: A Tutorial and Critique of a Paper by Hoare and Lauer, April 1979, AD A068-967
TM-131	Adleman, Leonard M. Time, Space and Randomness, April 1979
TM-132	Patil, Ramesh S. Design of a Program for Expert Diagnosis of Acid Base and Electrolyte Disturbances, May 1979

TM-133	Loui, Michael C. The Space Complexity of Two Pebble Games on Trees, May 1979
TM-134	Shamir, Adi How to Share a Secret, May 1979, AD A069-397
TM-135	Wyleczuk, Rosanne H. Timestamps and Capability-Based Protection in a Distributed Computer Facility, S.B. & S.M. Thesis, EE & CS Dept., June 1979
TM-136	Misunas, David P. Report on the Second Workshop on Data Flow Computer and Program Organization, June 1979
TM-137	Davis, Ernest and Jeffrey M. Jaffe Algorithms for Scheduling Tasks on Unrelated Processors, June 1979
TM-138	Pratt, Vaughan R. Dynamic Algebras: Examples, Constructions, Applications, July 1979
TM-139	Martin, William A. Roles, Co-Descriptors, and the Formal Representation of Quantified English Expressions (Revised May 1980), September 1979, AD A074-625
TM-140	Szolovits, Peter Artificial Intelligence and Clinical Problem Solving, September 1979
TM-141	Hammer, Michael and Dennis McLeod On Database Management System Architecture, October 1979, AD A076-417
TM-142	Lipski, Witold, Jr. On Data Bases with Incomplete Information, October 1979
TM-143	Leth, James W. An Intermediate Form for Data Flow Programs, S.M. Thesis, EE & CS Dept., November 1979
TM-144	Takagi, Akihiro Concurrent and Reliable Updates of Distributed Databases, November 1979

TM-145	Loui, Michael C. A Space Bound for One-Tape Multidimensional Turing Machines, November 1979
TM-146	Aoki, Donald J. A Machine Language Instruction Set for a Data Flow Processor, S.M. Thesis, EE & CS Dept., December 1979
TM-147	Schroeppel, Richard and Adi Shamir A $T = 0(2^{n/2})$, $S = 0(2^{n/4})$ Algorithm for Certain NP-Complete Problems, January 1980, AD A080-385
TM 148	Adleman, Leonard M. and Michael C. Loui Space-Bounded Simulation of Multitape Turing Machines, January 1980
TM-149	Pallottino, Stefano and Tommaso Toffoli An Efficient Algorithm for Determining the Length of the Longest Dead Path in an "Lifo" Branch-and-Bound Exploration Schema, January 1980, AD A079-912
TM-150	Meyer, Albert R. Ten Thousand and One Logics of Programming, February 1980
TM-151	Toffoli, Tommaso Reversible Computing, February 1980, AD A082-021
TM-152	Papadimitriou, Christos H. On the Complexity of Integer Programming, February 1980
TM-153	Papadimitriou, Christos H. Worst-Case and Probabilistic Analysis of a Geometric Location Problem, February 1980
TM-154	Karp, Richard M. and Christos H. Papadimitriou On Linear Characterizations of Combinatorial Optimization Problems, February 1980
TM-155	Atai, Alon, Richard J. Lipton, Christos H. Papadimitriou and M. Rodeh Covering Graphs by Simple Circuits, February 1980
TM-156	Meyer, Albert R. and Rohit Parikh Definability in Dynamic Logic, February 1980

PUBLICATIONS

TM-157	Meyer, Albert R. and Karl Winklmann On the Expressive Power of Dynamic Logic, February 1980
TM-158	Stark, Eugene W. Semaphore Primitives and Starvation-Free Mutual Exclusion, S.M. Thesis, EE & CS Dept., March 1980
TM-159	Pratt, Vaughan R. *Dynamic Algebras and the Nature of Induction*, March 1980
TM-160	Kanellakis, Paris C. On the Computational Complexity of Cardinality Constraints in Relational Databases, March 1980
TM-161	Lloyd, Errol L. Critical Path Scheduling of Task Systems with Resource and Processor Constraints, March 1980
TM-162	Marcum, Alan M. A Manager for Named, Permanent Objects, S.B. & S.M. Thesis, EE & CS Dept., April 1980, AD A083-491
TM-163	Meyer, Albert R. and Joseph Y. Halpern Axiomatic Definitions of Programming Languages: A Theoretical Assessment, April 1980
TM-164	Shamir, Adi The Cryptographic Security of Compact Knapsacks (Preliminary Report), April 1980, AD A084-456
TM-165	Finseth, Craig A. Theory and Practice of Text Editors or A Cookbook for an Emacs, S.B. Thesis, EE & CS Dept., May 1980
TM-166	Bryant, Randal E. Report on the Workshop on Self-Timed Systems, May 1980
TM-167	Pavelle, Richard and Michael Wester Computer Programs for Research in Gravitation and Differential Geometry, June 1980
TM-168	Greif, Irene Programs for Distributed Computing: The Calendar Application, July 1980, AD A087-357

TM-169	Burke, Glenn and David Moon LOOP Iteration Macro, (revised January 198l) July 1980, AD A087-372
TM-170	Ehrenfeucht, Andrzej, Rohit Parikh and Gregorz Rozenberg Pumping Lemmas for Regular Sets, August 1980
TM-171	Meyer, Albert R. What is a Model of the Lambda Calculus?, August 1980
TM-172	Paseman, William G. Some New Methods of Music Synthesis, S.M. Thesis, EE & CS Dept., August 1980, AD A090-130
TM-173	Hawkinson, Lowell B. XLMS: A Linguistic Memory System, September 1980, AD A090-033
TM-174	Arvind, Vinod Kathail and Keshav Pingali A Dataflow Architecture with Tagged Tokens, September 1980
TM-175	Meyer, Albert R., Daniel Weise and Michael C. Loui On Time Versus Space III, September 1980
TM-176	Seaquist, Carl R. A Semantics of Synchronization, S.M. Thesis, EE & CS Dept., September 1980, AD A091-015
TM-177	Sinha, Mukul K. TIMEPAD - A Performance Improving Synchronization Mechanism for Distributed Systems, September 1980
TM-178	Arvind and Robert E. Thomas I-Structures: An Efficient Data Type for Functional Languages, September 1980
TM-179	Halpern Joseph Y. and Albert R. Meyer Axiomatic Definitions of Programming Languages, II, October 1980
TM-180	Papadimitriou, Christos H. A Theorem in Database Concurrency Control, October 1980
TM-181	Lipski, Witold Jr. and Christos H. Papadimitriou A Fast Algorithm for Testing for Safety and Detecting Deadlocks in Locked Transaction Systems, October 1980

TM-182	Itai, Alon, Christos H. Papadimitriou and Jayme Luiz Szwarefiter Hamilton Paths in Grid Graphs, October 1980
TM-183	Meyer, Albert R. A Note on the Length of Craig's Interpolants, October 1980
TM-184	Lieberman, Henry and Carl Hewitt A Real Time Garbage Collector that can Recover Temporary Storage Quickly, October 1980
TM-185	Kung, Hsing-Tsung and Christos H. Papadimitriou An Optimality Theory of Concurrency Control for Databases, November 1980, AD A092-625
TM-186	Szolovits, Peter and William A. Martin BRAND X Manual, November 1980, AD A093-041
TM-187	Fischer, Michael J., Albert R. Meyer and Michael S. Paterson CapOmega()(n log n) Lower Bounds on Length of Boolean Formulas, November 1980
TM-188	Mayr, Ernst An Effective Representation of the Reachability Set of Persistent Petri Nets, January 1981
TM-189	Mayr, Ernst Persistence of Vector Replacement Systems is Decidable, January 1981
TM-190	Ben-Ari, Mordechai, Joseph Y. Halpern and Amir Pnueli Deterministic Propositional Dynamic Logic: Finite Models, Comple :ity, and Completeness, January 1981.
TM-191	Parikh, Rohit Propositional Dynamic Logics of Programs: A Survey, January 198l.
TM-192	Meyer, Albert R., Robert S. Streett and Grazina Mirkowska The Deducibility Problem in Propositional Dynamic Logic, February 1981
TM-193	Yannakakis, Mihalis and Christos H. Papadimitriou Algebraic Dependencies, February 1981

TM-194	Barendregt, Henk and Giuseppe Longo Recursion Theoretic Operators and Morphisms on Numbered Sets, February 1981
TM-195	Barber, Gerald R. Record of the Workshop on Research in Office Semantics, February 1981
TM-196	Bhatt, Sandeep N. On Concentration and Connection Networks, S.M. Thesis, EE & CS Dept., March 1981
TM-197	Fredkin, Edward and Toffoli Thomaso Conservative Logic, May 1981
TM-198	Halpern, Joseph and Reif, J. The Propositional Dynamic Logic of Deterministic Well-Structured Programs, March 1981
TM-199	Mayr, E. and Meyer, A. The Complexity of the Word Problems for Communative Semigroups and Polynomial Ideals, June 1981
TM-200	Burke, G. LSB Manual, June 1981
TM-201	Meyer, A. What is a Model of the lambda Calculus? Expanded Version, July 1981.
TM-202	Saltzer, J. H. Communication Ring Initialization without Central Control December 1981
TM-203	Bawden, A., Burke, G. and Hoffman, C. Maclisp Extensions, July 1981
TM-204	Halpern, J.Y. On the Expressive Power of Dynamic Logic, II, August 1981
TM-205	Kannon, R. Circuit-Size Lower Bounds and Non-Reducibility to Sparce Sets, October 1981.
TM-206	Leiserson, C. and Pinter, R. Optimal Placement for River Routing, October 1981

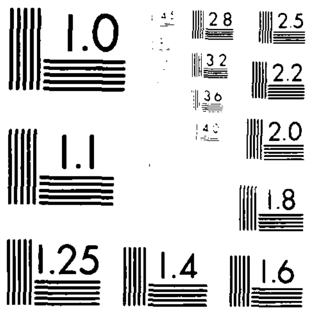
TM-207	Longo, G. Power Set Models For Lambda-Calculus: Theories, Expansions, Isomorphisms, November 1981
TM-208	Cosmadakis, S and Papadimitriou, C. The Traveling Salesman Problem with Many Visits to Few Cities, November 1981
TM-209	Johnson, D. and Papadimitriou, C. Computational Complexity and the Traveling Salesman Problem, December 1981
TM-210	Greif, I. Software for the 'Roiels" People Play, February 1982
TM-211	Meyer, A. and Tiuryn, J. A Note on Equivalences Among Logics of Programs, December 1981
TM-212	Elias, P. Minimax Optimal Universal Codeword Sets, January 1982
TM-213	Greif, I PCAL: A Personal Calendar, January 1982
TM-214	Meyer, A. and Mitchell, J. Terminations for Recursive Programs: Completeness and Axiomatic Definability, March 1982
TM-215	Leiserson, C. and Saxe J. Optimizing Synchronous Systems, March 1982
TM-216	Church, K. and Patil, R. Coping with Syntactic Ambiguity or How to Put the Block in the Box on the Table, April 1982.
TM-217	Wright, D. A File Transfer Program for a Personal Computer, April 1982
TM-218	Greif, I. Cooperative Office Work, Teleconferencing and Calendar Management: A Collection of Papers, May 1982
TM-219	Jouannaud, J.-P., Lescanne, P and Reinig, F. Recursive Decomposition Ordering and Multiset Orderings, June 1982
TM-220	Chu, T.-A. Circuit Analysis of Self-Times Elements for NMOS VLSI Systems, May 1982
TM-221	Leighton, F., Lepley, M. and Miller, G. Layouts for the Shuffle-Exchange Graph Based on the Complex Plane Diagram, June 1982
TM-222	Meier zu Sieker, F. A Telex Gateway for the Internety, S.B. Thesis, Electrical Engineering Dept., May 1982

AD-A145 134 MIT LABORATORY FOR COMPUTER SCIENCE PROGRESS REPORT 20 - JULY 1982 - JUN 1983(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE M L DERTOUZOS

3/3

UNCLASSIFIED JUL 84 N00014-83-K-0125 F/G 9/2 NL

END
DATE
FILMED
10-84
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

TM-223	diSessa, A.A. A Principled Design for an Integrated Computation Environment, July 1982
TM-224	Barber, G. Supporting Organizational Problem Solving with a Workstation, July 1982
TM-225	Barber, G. and Hewitt, C. Foundations for Office Semantics, July 1982
TM-226	Bergstra, J., Chmielinska, A. and Tiuryn, J. Hoares's Logic Not Complete When it Could Be, August 1982
TM-227	Leighton, F.T. New Lower Bound Techniques for VLSI, August 1982
TM-228	Papadimitriou, C. and Zachos, S. Two Remarks on the Power of Counting, August 1982
TM-229	Cosmadakis, S. The Complexity of Evaluation Relational Queries, August 1982
TM-230	Shamir, A. Embedding Cryptographic Trapdoors in Arbitrary Knapsack Systems, September 1982
TM-231	Kleitman, D., Leighton, F.T., Lepley, M. and Miller G. An Asymptotically Optimal Layout for the shuffle-exchange Graph, October 1982
TM-232	Yeh, A. PLY: A System of Plausibility Inference with a Probabilistic Basis, December 1982
TM-233	Konopelski, L. Implementing Internet Remote Login on a Personal Computer, S.B. Thesis, Electrical Engineering Dept., December 1982
TM-234	Rivest, R. and Sherman, A. Randomized Encryption Techniques, January 1983.
TM-235	Mitchell, J. The Implication of Problem for Functional and Inclusion Dependencies, February 1983
TM-236	Leighton,. F.T. and Leiserson, C.E. Wafter-Scale Integration of Systolic Arrays, February 1983
TM-237	Dolev, D., Leighton, F.T. and Trickey, H. Planar Embedding of Planar Graphs, February 1983

PUBLICATIONS

TM-238	Baker, B.S., Bhatt, S.N. and Leighton, F.T. An Approximation Algorithm for Manhattan Routing, February 1983
TM-239	Sutherland, J.B. and Sirbu, M. Evaluation of an Office Analysis Methodology, March 1983
TM-240	Bromley, H. A Program for Therapy of Acid-Base and Electrolyte Disorders, S.B. Thesis, Electrical Engineering Dept., June 1983.

Technical Reports

TR-1[10]	Bobrow, Daniel G. Natural Language Input for a Computer Problem Solving System, Ph.D. Dissertation, Math. Dept., September 1964, AD 604-730
TR-2	Raphael, Bertram SIR: A Computer Program for Semantic Information Retrieval, Ph.D. Dissertation, Math. Dept., June 1964, AD 608-499
TR-3	Corbato, Fernando J. System Requirements for Multiple-Access, Time-Shared Computers, May 1964, AD 608-501
TR-4	Ross, Douglas T. and Clarence G. Feldman Verbal and Graphical Language for the AED System: A Progress Report, May 1964, AD 604-678
TR-6	Biggs, John M. and Robert D. Logcher STRESS: A Problem-Oriented Language for Structural Engineering, May 1964, AD 604-679
TR-7	Weizenbaum, Joseph OPL-1: An Open Ended Programming System within CTSS, April 1964, AD 604-680
TR-8	Greenberger, Martin The OPS-1 Manual, May 1964, AD 604-681
TR-11	Dennis, Jack B. Program Structure in a Multi-Access Computer, May 1964, AD 608-500
TR-12	Fano, Robert M. The MAC System: A Progress Report, October 1964, AD 609-296
TR-13	Greenberger, Martin A New Methodology for Computer Simulation, October 1964, AD 609-288

[10] TRs 5, 9, 10, 15 were never issued

TR-14	Roos, Daniel Use of CTSS in a Teaching Environment, November 1964, AD 661-807
TR-16	Saltzer, Jerome H. CTSS Technical Notes, March 1965, AD 612-702
TR-17	Samuel, Arthur L. Time-Sharing on a Multiconsole Computer, March 1965, AD 462-158
TR-18	Scherr, Allan Lee An Analysis of Time-Shared Computer Systems, Ph.D. Dissertation, EE Dept., June 1965, AD 470-715
TR-19	Russo, Francis John A Heuristic Approach to Alternate Routing in a Job Shop, S.B. & S.M. Thesis, Sloan School, June 1965, AD 474-018
TR-20	Wantman, Mayer Elihu CALCULAID: An On-Line System for Algebraic Computation and Analysis, S.M. Thesis, Sloan School, September 1965, AD 474-019
TR-21	Denning, Peter James Queueing Models for File Memory Operation, S.M. Thesis, EE Dept., October 1965, AD 624-943
TR-22	Greenberger, Martin The Priority Problem, November 1965, AD 625-728
TR-23	Dennis, Jack B. and Earl C. Van Horn Programming Semantics for Multi-programmed Computations, December 1965, AD 627-537
TR-24	Kaplow, Roy, Stephen Strong and John Brackett MAP: A System for On-Line Mathematical Analysis, January 1966, AD 476-443
TR-25	Stratton, William David Investigation of an Analog Technique to Decrease Pen-Tracking Time in Computer Displays, S.M. Thesis, EE Dept., March 1966, AD 631-396

TR-26	Cheek, Thomas Burrell Design of a Low-Cost Character Generator for Remote Computer Displays, S.M. Thesis, EE Dept., March 1966, AD 631-269
TR-27	Edwards, Daniel James OCAS - On-Line Cryptanalytic Aid System, S.M. Thesis, EE Dept., May 1966, AD 633-678
TR-28	Smith, Arthur Anshel Input/Output in Time-Shared, Segmented, Multiprocessor Systems, S.M. Thesis, EE Dept., June 1966, AD 637-215
TR-29	Ivie, Evan Leon Search Procedures Based on Measures of Relatedness between Documents, Ph.D. Dissertation, EE Dept., June 1966, AD 636-275
TR-30	Saltzer, Jerome Howard TRaffic Control in a Multiplexed Computer System, Sc.D. Thesis, EE Dept., July 1966, AD 635-966
TR-31	Smith, Donald L. Models and Data Structures for Digital Logic Simulation, S.M. Thesis, EE Dept., August 1966, AD 637-192
TR-32	Teitelman, Warren PILOT: A Step Toward Man-Computer Symbiosis, Ph.D. Dissertation, Math. Dept., September 1966, AD 638-446
TR-33	Norton, Lewis M, ADEPT - A Heuristic Program for Proving Theorems of Group Theory, Ph.D. Dissertation, Math. Dept., October 1966, AD 645-660
TR-34	Van Horn, Earl C., Jr. Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. Dissertation, EE Dept., November 1966, AD 650-407
TR-35	Fenichel, Robert R. An On-Line System for Algebraic Manipulation, Ph.D. Dissertation, Appl. Math. (Harvard), December 1966, AD 657-282
TR-36	Martin, William A. Symbolic Mathematical Laboratory, Ph.D. Dissertation, EE Dept., January 1967, AD 657-283
TR-37	Guzman-Arenas, Adolfo

	Some Aspects of Pattern Recognition by Computer, S.M. Thesis, EE Dept., February 1967, AD 656-041
TR-38	Rosenberg, Ronald C., Daniel W. Kennedy and Roger A. Humphrey A Low-Cost Output Terminal For Time-Shared Computers, March 1967, AD 662-027
TR-39	Forte, Allen Syntax-Based Analytic Reading of Musical Scores, April 1967, AD 661-806
TR-40	Miller, James R. On-Line Analysis for Social Scientists, May 1967, AD 668-009
TR-41	Coons, Steven A. Surfaces for Computer-Aided Design of Space Forms, June 1967, AD 663-504
TR-42	Liu, Chung L., Gabriel D. Chang and Richard E. Marks Design and Implementation of a Table-Driven Compiler System, July 1967, AD 668-960
TR-43	Wilde, Daniel U. Program Analysis by Digital Computer, Ph.D. Dissertation, EE Dept., August 1967, AD 662-224
TR-44	Gorry, G. Anthony A System for Computer-Aided Diagnosis, Ph.D. Dissertation, Sloan School, September 1967, AD 662-665
TR-45	Leal-Cantu, Nestor On the Simulation of Dynamic Systems with Lumped Parameters and Time Delays, S.M. Thesis, ME Dept., October 1967, AD 663-502
TR-46	Alsop, Joseph W. A Canonic Translator, S.B. Thesis, EE Dept., November 1967, AD 663-503
TR-47	Moses, Joel Symbolic Integration, Ph.D. Dissertation, Math. Dept., December 1967, AD 662-666
TR-48	Jones, Malcolm M.

Incremental Simulation on a Time-Shared Computer, Ph.D. Dissertation, Sloan School, January 1968, AD 662-225

TR-49 Luconi, Fred L.
Asynchronous Computational Structures, Ph.D. Dissertation, EE Dept., February 1968, AD 667-602

TR-50 Denning, Peter J.
Resource Allocation in Multiprocess Computer Systems, Ph.D. Dissertation, EE Dept., May 1968, AD 675-554

TR-51 Charniak, Eugene
CARPS, A Program which Solves Calculus Word Problems, S.M. Thesis, EE Dept., July 1968, AD 673-670

TR-52 Deitel, Harvey M.
Absentee Computations in a Multiple-Access Computer System, S.M. Thesis, EE Dept., August 1968, AD 684-738

TR-53 Slutz, Donald R.
The Flow Graph Schemata Model of Parallel Computation, Ph.D. Dissertation, EE Dept., September 1968, AD 683-393

TR-54 Grochow, Jerrold M.
The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System, S.M. Thesis, EE Dept., October 1968, AD 689-468

TR-55 Rappaport, Robert L.
Implementing Multi-Process Primitives in a Multiplexed Computer System, S.M. Thesis, EE Dept., November 1968, AD 689-469

TR-56 Thornhill, Daniel E., Robert H. Stotz, Douglas T. Ross and John E. Ward
An Integrated Hardware-Software System for Computer Graphics in Time-Sharing, December 1968, AD 685-202

TR-57 Morris, James H.
Lambda-Calculus Models of Programming Languages, Ph.D. Dissertation, Sloan School, December 1968, AD 683-394

TR-58 Greenbaum, Howard J.
A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System, S.M. Thesis, EE Dept., January 1969, AD 686-988

TR-59 Guzman, Adolfo
Computer Recognition of Three- Dimensional Objects in a Visual Scene, Ph.D. Dissertation, EE Dept., December 1968, AD 692-200

TR-60 Ledgard, Henry F.
A Formal System for Defining the Syntax and Semantics of Computer Languages, Ph.D. Dissertation, EE Dept., April 1969. AD 689-305

TR-61 Baecker, Ronald M.
Interactive Computer-Mediated Animation, Ph.D. Dissertation, EE Dept., June 1969, AD 690-887

TR-62 Tillman, Coyt C., Jr.
EPS: An Interactive System for Solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation, June 1969, AD 692-462

TR-63 Brackett, John W., Michael Hammer and Daniel E. Thornhill
Case Study in Interactive Graphics Programming: A Circuit Drawing and Editing Program for Use with a Storage-Tube Display Terminal, October 1969, AD 699-930

TR-64 Rodriguez, Jorge E.
A Graph Model for Parallel Computations, Sc.D. Thesis, EE Dept., September 1969, AD 697-759

TR-65 DeRemer, Franklin L.
Practical Translators for LR(k) Languages, Ph.D. Dissertation, EE Dept., October 1969, AD 699-501

TR-66 Beyer, Wendell T.
Recognition of Topological Invariants by Iterative Arrays, Ph.D. Dissertation, Math. Dept., October 1969, AD 699-502

TR-67	Vanderbilt, Dean H. Controlled Information Sharing in a Computer Utility, Ph.D. Dissertation, EE Dept., October 1969, AD 699-503
TR-68	Selwyn, Lee L. Economies of Scale in Computer Use: Initial Tests and Implications for The Computer Utility, Ph.D. Dissertation, Sloan School, June 1970, AD 710-011
TR-69	Gertz, Jeffrey L. Hierarchical Associative Memories for Parallel Computation, Ph.D. Dissertation, EE Dept., June 1970, AD 711-091
TR-70	Fillat, Andrew I. and Leslie A. Kraning Generalized Organization of Large Data-Bases: A Set-Theoretic Approach to Relations, S.B. & S.M. Thesis, EE Dept., June 1970, AD 711-060
TR-71	Fiasconaro, James G. A Computer-Controlled Graphical Display Processor, S.M. Thesis, EE Dept., June 1970, AD 710-479
TR-72	Patil, Suhas S. Coordination of Asynchronous Events, Sc.D. Thesis, EE Dept., June 1970, AD 711-763
TR-73	Griffith, Arnold K. Computer Recognition of Prismatic Solids, Ph.D. Dissertation, Math. Dept., August 1970, AD 712-069
TR-74	Edelberg, Murray Integral Convex Polyhedra and an Approach to Integralization, Ph.D. Dissertation, EE Dept., August 1970, AD 712-070
TR-75	Hebalkar, Prakash G. Deadlock-Free Sharing of Resources in Asynchronous Systems, Sc.D. Thesis, EE Dept., September 1970, AD 713-139
TR-76	Winston, Patrick H. Learning Structural Descriptions from Examples, Ph.D. Dissertation, EE Dept., September 1970, AD 713-988
TR-77	Haggerty, Joseph P. Complexity Measures for Language Recognition by Canonic Systems, S.M. Thesis, EE Dept., October 1970, AD 715-134

TR-78	Madnick, Stuart E. Design Strategies for File Systems, S.M. Thesis, EE Dept. & Sloan School, October 1970, AD 714-269
TR-79	Horn, Berthold K. Shape from Shading: A Method for Obtaining the Shape of a Smooth Opaque Object from One View, Ph.D. Dissertation, EE Dept., November 1970, AD 717-336
TR-80	Clark, David D., Robert M. Graham, Jerome H. Saltzer and Michael D. Schroeder The Classroom Information and Computing Service, January 1971, AD 717-857
TR-81	Banks, Edwin R. Information Processing and Transmission in Cellular Automata, Ph.D. Dissertation, ME Dept., January 1971, AD 717-951
TR-82	Krakauer, Lawrence J. Computer Analysis of Visual Properties of Curved Objects, Ph.D. Dissertation, EE Dept., May 1971, AD 723-647
TR-83	Lewin, Donald E. In-Process Manufacturing Quality Control, Ph.D. Dissertation, Sloan School, January 1971, AD 720-098
TR-84	Winograd, Terry Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, Ph.D. Dissertation, Math. Dept., February 1971, AD 721-399
TR-85	Miller, Perry L. Automatic Creation of a Code Generator from a Machine Description, E.E. Thesis, EE Dept., May 1971, AD 724-730
TR-86	Schell, Roger R. Dynamic Reconfiguration in a Modular Computer System, Ph.D. Dissertation, EE Dept., June 1971, AD 725-859
TR-87	Thomas, Robert H. A Model for Process Representation and Synthesis, Ph.D. Dissertation, EE Dept., June 1971, AD 726-049

TR-88	Welch, Terry A. Bounds on Information Retrieval Efficiency in Static File Structures, Ph.D. Dissertation, EE Dept., June 1971, AD 725-429
TR-89	Owens, Richard C., Jr. Primary Access Control in Large-Scale Time-Shared Decision Systems, S.M. Thesis, Sloan School, July 1971, AD 728-036
TR-90	Lester, Bruce P. Cost Analysis of Debugging Systems, S.B. & S.M. Thesis, EE Dept., September 1971, AD 730-521
TR-91	Smoliar, Stephen W. A Parallel Processing Model of Musical Str[illegible]res, Ph.D. Dissertation, Math. Dept., September 1971, AD 7[illegible]590
TR-92	Wang, Paul S. Evaluation of Definite Integrals by Symbolic Ma[illegible]tion, Ph.D. Dissertation, Math. Dept., October 1971, AD 732-[illegible]
TR-93	Greif, irene Gloria *Induction in Proofs about Programs*, S.M. Thesis, EE Dept., February 1972, AD 737-701
TR-94	Hack, Michel Henri Theodore Analysis of Production Schemata by Petri Nets, S.M. Thesis, EE Dept., February 1972, AD 740-320
TR-95	Fateman, Richard J. Essays in Algebraic Simplification (A revision of a Harvard Ph.D. Dissertation), April 1972, AD 740-132
TR-96	Manning, Frank Autonomous, Synchronous Counters Constructed Only of J-K Flip-Flops, S.M. Thesis, EE Dept., May 1972, AD 744-030
TR-97	Vilfan, Bostjan The Complexity of Finite Functions, Ph.D. Dissertation, EE Dept., March 1972, AD 739-678
TR-98	Stockmeyer, Larry Joseph Bounds on Polynomial Evaluation Algorithms, S.M. Thesis, EE Dept., April 1972, AD 740-328

PUBLICATIONS

TR-99	Lynch, Nancy Ann Relativization of the Theory of Computational Complexity, Ph.D. Dissertation, Math. Dept., June 1972, AD 744-032
TR-100	Mandl, Robert Further Results on Hierarchies of Canonic Systems, S.M. Thesis, EE Dept., June 1972, AD 744-206
TR-101	Dennis, Jack B. On the Design and Specification of a Common Base Language, June 1972, AD 744-207
TR-102	Hossley, Robert F. Finite Tree Automata and ω-Automata, S.M. Thesis, EE Dept., September 1972, AD 749-367
TR-103	Sekino, Akira Performance Evaluation of Multiprogrammed Time-Shared Computer Systems, Ph.D. Dissertation, EE Dept., September 1972, AD 749-949
TR-104	Schroeder, Michael D. Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. Dissertation, EE Dept., September 1972, AD 750-173
TR-105	Smith, Burton J. An Analysis of Sorting Networks, Sc.D. Thesis, EE Dept., October 1972, AD 751-614
TR-106	Rackoff, Charles W. The Emptiness and Complementation Problems for Automata on Infinite Trees, S.M. Thesis, EE Dept., January 1973, AD 756-248
TR-107	Madnick, Stuart E. Storage Hierarchy Systems, Ph.D. Dissertation, EE Dept., April 1973, AD 760-001
TR-108	Wand, Mitchell Mathematical Foundations of Formal Language Theory, Ph.D. Dissertation, Math. Dept., December 1973.
TR-109	Johnson, David S. Near-Optimal Bin Packing Algorithms, Ph.D. Dissertation, Math. Dept., June 1973, PB 222-090

TR-110	Moll, Robert Complexity Classes of Recursive Functions, Ph.D. Dissertation, Math. Dept., June 1973, AD 767-730
TR-111	Linderman, John P. Productivity in Parallel Computation Schemata, Ph.D. Dissertation, EE Dept., December 1973, PB 226-159/AS
TR-112	Hawryszkiewycz, Igor T. Semantics of Data Base Systems, Ph.D. Dissertation, EE Dept., December 1973, PB 226-061/AS
TR-113	Herrmann, Paul P. On Reducibility Among Combinatorial Problems, S.M. Thesis, Math. Dept., December 1973, PB 226-157/AS
TR-114	Metcalfe, Robert M. Packet Communication, Ph.D. Dissertation, Applied Math., Harvard University, December 1973, AD 771-430
TR-115	Rotenberg, Leo *Making Computers Keep Secrets*, Ph.D. Dissertation, EE Dept., February 1974, PB 229-352/AS
TR-116	Stern, Jerry A. Backup and Recovery of On-Line Information in a Computer Utility, S.M. & E.E. Thesis, EE Dept., January 1974, AD 774-141
TR-117	Clark, David D. An Input/Output Architecture for Virtual Memory Computer Systems, Ph.D. Dissertation, EE Dept., January 1974, AD 774-738
TR-118	Briabrin, Victor An Abstract Model of a Research Institute: Simple Automatic Programming Approach, March 1974, PB 231-505/AS
TR-119	Hammer, Michael M. A New Grammatical Transformation into Deterministic Top-Down Form, Ph.D. Dissertation, EE Dept., February 1974, AD 775-545
TR-120	Ramchandani, Chander Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Ph.D. Dissertation, EE Dept., February 1974, AD 775-618

TR-121 Yao, Foong F.
On Lower Bounds for Selection Problems, Ph.D. Dissertation, Math. Dept., March 1974, PB 230-950/AS

TR-122 Scherf, John A.
Computer and Data Security: A Comprehensive Annotated Bibliography, S.M. Thesis, Sloan School, January 1974, AD 775-546

TR-123 Introduction to Multics
February 1974, AD 918-562

TR-124 Laventhal, Mark S.
Verification of Programs Operating on Structured Data, S.B. & S.M. Thesis, EE Dept., March 1974, PB 231-365/AS

TR-125 Mark, William S.
A Model-Debugging System, S.B. & S.M. Thesis, EE Dept., April 1974, AD 778-688

TR-126 Altman, Vernon E.
A Language Implementation System, S.B. & S.M. Thesis, Sloan School, May 1974, AD 780-672

TR-127 Greenberg, Bernard S.
An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory, S.M. Thesis, EE Dept., May 1974, AD 780-407

TR-128 Frankston, Robert M.
The Computer Utility as a Marketplace for Computer Services, S.M. & E.E. Thesis, EE Dept., May 1974, AD 780-436

TR-129 Weissberg, Richard W.
Using Interactive Graphics in Simulating the Hospital Emergency Room, S.M. Thesis, EE Dept., May 1974, AD 780-437

TR-130 Ruth, Gregory R.
Analysis of Algorithm Implementations, Ph.D. Dissertation, EE Dept., May 1974, AD 780-408

TR-131 Levin, Michael
Mathematical Logic for Computer Scientists, June 1974.

TR-132	Janson, Philippe A. Removing the Dynamic Linker from the Security Kernel of a Computing Utility, S.M. Thesis, EE Dept., June 1974, AD 781-305
TR-133	Stockmeyer, Larry J. The Complexity of Decision Problems in Automata Theory and Logic, Ph.D. Dissertation, EE Dept., July 1974, PB 235-283/AS
TR-134	Ellis, David J. Semantics of Data Structures and References, S.M. & E.E. Thesis, EE Dept., August 1974, PB 236-594/AS
TR-135	Pfister, Gregory F. The Computer Control of Changing Pictures, Ph.D. Dissertation, EE Dept., September 1974, AD 787-795
TR-136	Ward, Stephen A. Functional Domains of Applicative Languages, Ph.D. Dissertation, EE Dept., September 1974, AD 787-796
TR-137	Seiferas, Joel I. Nondeterministic Time and Space Complexity Classes, Ph.D. Dissertation, Math. Dept., September 1974. PB 236-777/AS
TR-138	Yun, David Y. Y. The Hensel Lemma in Algebraic Manipulation, Ph.D. Dissertation, Math. Dept., November 1974, AD A002-737
TR-139	Ferrante, Jeanne Some Upper and Lower Bounds on Decision Procedures in Logic, Ph.D. Dissertation, Math. Dept., November 1974. PB 238-121/AS
TR-140	Redell, David D. Naming and Protection in Extendable Operating Systems, Ph.D. Dissertation, EE Dept., November 1974, AD A001-721
TR-141	Richards, Martin, A. Evans and R. Mabee The BCPL Reference Manual, December 1974, AD A003-599
TR-142	Brown, Gretchen P. Some Problems in German to English Machine Translation, S.M. & E.E. Thesis, EE Dept., December 1974, AD A003-002

TR-143	Silverman, Howard A Digitalis Therapy Advisor, S.M. Thesis, EE Dept., January 1975.
TR-144	Rackoff, Charles The Computational Complexity of Some Logical Theories, Ph.D. Dissertation, EE Dept., February 1975.
TR-145	Henderson, D. Austin The Binding Model: A Semantic Base for Modular Programming Systems, Ph.D. Dissertation, EE Dept., February 1975, AD A006-961
TR-146	Malhotra, Ashok Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis, Ph.D. Dissertation, EE Dept., February 1975.
TR-147	Van De Vanter, Michael L. A Formalization and Correctness Proof of the CGOL Language System, S.M. Thesis, EE Dept., March 1975.
TR-148	Johnson, Jerry *Program Restructuring for* Virtual Memory Systems, Ph.D. Dissertation, EE Dept., March 1975, AD A009-218
TR-149	Snyder, Alan A Portable Compiler for the Language C, S.B. & S.M. Thesis, EE Dept., May 1975, AD A010-218
TR-150	Rumbaugh, James E. A Parallel Asynchronous Computer Architecture for Data Flow Programs, Ph.D. Dissertation, EE Dept., May 1975, AD A010-918
TR-151	Manning, Frank B. Automatic Test, Configuration, and Repair of Cellular Arrays, Ph.D. Dissertation, EE Dept., June 1975, AD A012-822
TR-152	Qualitz, Joseph E. Equivalence Problems for Monadic Schemas, Ph.D. Dissertation, EE Dept., June 1975, AD A012-823
TR-153	Miller, Peter B. Strategy Selection in Medical Diagnosis, S.M. Thesis, EE & CS Dept., September 1975.

TR-154 Greif, Irene
Semantics of Communicating Parallel Processes, Ph.D. Dissertation, EE & CS Dept., September 1975, AD A016-302

TR-155 Kahn, Kenneth M.
Mechanization of Temporal Knowledge, S.M. Thesis, EE & CS Dept., September 1975.

TR-156 Bratt, Richard G.
Minimizing the Naming Facilities Requiring Protection in a Computer Utility, S.M. Thesis, EE & CS Dept., September 1975.

TR-157 Meldman, Jeffrey A.
A Preliminary Study in Computer-Aided Legal Analysis, Ph.D. Dissertation, EE & CS Dept., November 1975, AD A018-997

TR-158 Grossman, Richard W.
Some Data-base Applications of Constraint Expressions, S.M. Thesis, EE & CS Dept., February 1976, AD A024-149

TR-159 Hack, Michel
Petri Net Languages, March 1976.

TR-160 Bosyj, Michael
A Program for the Design of Procurement Systems, S.M. Thesis, EE & CS Dept., May 1976, AD A026-688

TR-161 Hack, Michel
Decidability Questions, Ph.D. Dissertation, EE & CS Dept., June 1976.

TR-162 Kent, Stephen T.
Encryption-Based Protection Protocols for Interactive User-Computer Communication, S.M. Thesis, EE & CS Dept., June 1976, AD A026-911

TR-163 Montgomery, Warren A.
A Secure and Flexible Model of Process Initiation for a Computer Utility, S.M. & E.E. Thesis, EE & CS Dept., June 1976.

TR-164 Reed, David P.
Processor Multiplexing in a Layered Operating System, S.M. Thesis, EE & CS Dept., July 1976.

TR-165	McLeod, Dennis J. High Level Expression of Semantic Integrity Specifications in a Relational Data Base System, S.M. Thesis, EE & CS Dept., September 1976, AD A034-184
TR-166	Chan, Arvola Y. Index Selection in a Self-Adaptive Relational Data Base Management System, S.M. Thesis, EE & CS Dept., September 1976, AD A034-185
TR-167	Janson, Philippe A. Using Type Extension to Organize Virtual Memory Mechanisms, Ph.D. Dissertation, EE & CS Dept., September 1976.
TR-168	Pratt, Vaughan R. Semantical Considerations on Floyd-Hoare Logic, September 1976.
TR-169	Safran, Charles, James F. Desforges and Philip N. Tsichlis Diagnostic Planning and Cancer Management, September 1976.
TR-170	Furtek, Frederick C. The Logic of Systems, Ph.D. Dissertation, EE & CS Dept., December 1976.
TR-171	Huber, Andrew R. A Multi-Process Design of a Paging System, S.M. & E.E. Thesis, EE & CS Dept., December 1976.
TR-172	Mark, William S. The Reformulation Model of Expertise, Ph.D. Dissertation, EE & CS Dept., December 1976, AD A035-397
TR-173	Goodman, Nathan Coordination of Parallel Processes in the Actor Model of Computation, S.M. Thesis, EE & CS Dept., December 1976.
TR-174	Hunt, Douglas H. A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem, S.M. & E.E. Thesis, EE & CS Dept., December 1976.
TR-175	Goldberg, Harold J. A Robust Environment for Program Development, S.M. Thesis, EE & CS Dept., February 1977.

TR-176	Swartout, William R. A Digitalis Therapy Advisor with Explanations, S.M. Thesis, EE & CS Dept., February 1977.
TR-177	Mason, Andrew H. A Layered Virtual Memory Manager, S.M. & E.E. Thesis, EE & CS Dept., May 1977.
TR-178	Bishop, Peter B. Computer Systems with a Very Large Address Space and Garbage Collection, Ph.D. Dissertation, EE & CS Dept., May 1977, AD A040-601
TR-179	Karger, Paul A. Non-Discretionary Access Control for Decentralized Computing Systems, S.M. Thesis, EE & CS Dept., May 1977, AD A040-804
TR-180	Luniewski, Allen W. A Simple and Flexible System Initialization Mechanism, S.M. & E.E. Thesis, EE & CS Dept., May 1977.
TR-181	Mayr, Ernst W. The Complexity of the Finite Containment Problem for Petri Nets, S.M. Thesis, EE & CS Dept., June 1977 .
TR-182	Brown, Gretchen P. A Framework for Processing Dialogue, June 1977, AD A042-370
TR-183	Jaffe, Jeffrey M. Semilinear Sets and Applications, S.M. Thesis, EE & CS Dept., July 1977.
TR-184	Levine, Paul H. Facilitating Interprocess Communication in a Heterogeneous Network Environment, S.B. & S.M. Thesis, EE & CS Dept., July 1977, AD A043-901
TR-185	Goldman, Barry Deadlock Detection in Computer Networks, S.B. & S.M. Thesis, EE & CS Dept., September 1977, AD A047-025
TR-186	Ackerman, William B. A Structure Memory for Data Flow Computers, S.M. Thesis, EE & CS Dept., September 1977, AD A047-026

TR-187	Long, William J. A Program Writer, Ph.D. Dissertation, EE & CS Dept., November 1977, AD A047-595
TR-188	Bryant, Randal E. Simulation of Packet Communication Architecture Computer Systems, S.M. Thesis, EE & CS Dept., November 1977, AD A048-290
TR-189	Ellis, David J. Formal Specifications for Packet Communication Systems, Ph.D. Dissertation, EE & CS Dept., November 1977, AD A048-980
TR-190	Moss, J. Eliot B. Abstract Data Types in Stack Based Languages, S.M. Thesis, EE & CS Dept., February 1978, AD A052-332
TR-191	Yonezawa, Akinori Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, Ph.D. Dissertation, EE & CS Dept., January 1978, AD A051-149
TR-192	Niamir, Bahram Attribute Partitioning in a Self-Adaptive Relational Database System, S.M. Thesis, EE & CS Dept., January 1978, AD A053-292
TR-193	Schaffert, J. Craig A Formal Definition of CLU, S.M. Thesis, EE & CS Dept., January 1978
TR-194	Hewitt, Carl and Henry Baker, Jr. Actors and Continuous Functionals, February 1978, AD A052-266
TR-195	Bruss, Anna R. On Time-Space Classes and Their Relation to the Theory of Real Addition, S.M. Thesis, EE & CS Dept., March 1978
TR-196	Schroeder, Michael D., David D. Clark, Jerome H. Saltzer and Douglas H. Wells Final Report of the Multics Kernel Design Project, March 1978
TR-197	Baker, Henry Jr. Actor Systems for Real-Time Computation, Ph.D. Dissertation, EE & CS Dept., March 1978, AD A053-328

TR-198	Halstead, Robert H., Jr. Multiple-Processor Implementation of Message-Passing Systems, S.M. Thesis, EE & CS Dept., April 1978, AD A054-009
TR-199	Terman, Christopher J. The Specification of Code Generation Algorithms, S.M. Thesis, EE & CS Dept., April 1978, AD A054-301
TR-200	Harel, David Logics of Programs: Axiomatics and Descriptive Power, Ph.D. Dissertation, EE & CS Dept., May 1978
TR-201	Scheifler, Robert W. A Denotational Semantics of CLU, S.M. Thesis, EE & CS Dept., June 1978
TR-202	Principato, Robert N., Jr. A Formalization of the State Machine Specification Technique, S.M. & E.E. Thesis, EE & CS Dept., July 1978
TR-203	Laventhal, Mark S. Synthesis of Synchronization Code for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1978, AD A058-232
TR-204	Teixeira, Thomas J. Real-Time Control Structures for Block Diagram Schemata, S.M. Thesis, EE & CS Dept., August 1978, AD A061-122
TR-205	Reed, David P. Naming and Synchronization in a Decentralized Computer System, Ph.D. Dissertation, EE & CS Dept., October 1978, AD A061-407
TR-206	Borkin, Sheldon A. Equivalence Properties of Semantic Data Models for Database Systems, Ph.D. Dissertation, EE & CS Dept., January 1979, AD A066-386
TR-207	Montgomery, Warren A. Robust Concurrency Control for a Distributed Information System, Ph.D. Dissertation, EE & CS Dept., January 1979, AD A066-996

TR-208	Krizan, Brock C. A Minicomputer Network Simulation System, S.B. & S.M. Thesis, EE & CS Dept., February 1979
TR-209	Snyder, Alan A Machine Architecture to Support an Object-Oriented Language, Ph.D. Dissertation, EE & CS Dept., March 1979, AD A068-111
TR-210	Papadimitriou, Christos H. Serializability of Concurrent Database Updates, March 1979
TR-211	Bloom, Toby Synchronization Mechanisms for Modular Programming Languages, S.M. Thesis, EE & CS Dept., April 1979, AD A069-819
TR-212	Rabin, Michael O. Digitalized Signatures and Public-Key Functions as Intractable as Factorization, March 1979
TR-213	Rabin, Michael O. Probabilistic Algorithms in Finite Fields, March 1979
TR-214	McLeod, Dennis A Semantic Data Base Model and Its Associated Structured User Interface, Ph.D. Dissertation, EE & CS Dept., March 1979, AD A068-112
TR-215	Svobodova, Liba, Barbara Liskov and David Clark Distributed Computer Systems: Structure and Semantics, April 1979, AD A070-286
TR-216	Myers, John M. Analysis of the SIMPLE Code for Dataflow Computation, June 1979
TR-217	Brown, Donna J. Storage and Access Costs for Implementations of Variable - Length Lists, Ph.D. Dissertation, EE & CS Dept., June 1979
TR-218	Ackerman, William B. and Jack B. Dennis VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual, June 1979, AD A072-394

TR-219	Sollins, Karen R. Copying Complex Structures in a Distributed System, S.M. Thesis, EE & CS Dept., July 1979, AD A072-441
TR-220	Kosinski, Paul R. Denotational Semantics of Determinate and Non-Determinate Data Flow Programs, Ph.D. Dissertation, EE & CS Dept., July 1979
TR-221	Berzins, Valdis A. Abstract Model Specifications for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1979
TR-222	Halstead, Robert H., Jr. Reference Tree Networks: Virtual Machine and Implementation, Ph.D. Dissertation, EE & CS Dept., September 1979, AD A076-570
TR-223	Brown, Gretchen P. Toward a Computational Theory of Indirect Speech Acts, October 1979, AD A077-065
TR-224	Isaman, David L. Data-Structuring Operations in Concurrent Computations, Ph.D. Dissertation, EE & CS Dept., October 1979
TR-225	Liskov, Barbara, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler and Alan Snyder CLU Reference Manual, October 1979, AD A077-018
TR-226	Reuveni, Asher The Event Based Language and Its Multiple Processor Implementations. Ph.D. Dissertation, EE & CS Dept., January 1980, AD A081-950
TR-227	Rosenberg, Ronni L. Incomprehensible Computer Systems: Knowledge Without Wisdom, S.M. Thesis, EE & CS Dept., January 1980
TR-228	Weng, Kung-Song An Abstract Implementation for a Generalized Data Flow Language, Ph.D. Dissertation, EE & CS Dept., January 1980

TR-229	Atkinson, Russell R. Automatic Verification of Serializers, Ph.D. Dissertation, EE & CS Dept., March 1980, AD A082-885
TR-230	Baratz, Alan E. The Complexity of the Maximum Network Flow Problem, S.M. Thesis, EE & CS Dept., March 1980
TR-231	Jaffe, Jeffrey M. Parallel Computation: Synchronization, Scheduling, and Schemes, Ph.D. Dissertation, EE & CS Dept., March 1980
TR-232	Luniewski, Allen W. The Architecture of an Object Based Personal Computer, Ph.D. Dissertation, EE & CS Dept., March 1980, AD A083-433
TR-233	Kaiser, Gail E. Automatic Extension of an Augmented Transition Network Grammar for Morse Code Conversations, S.B. Thesis, EE & CS Dept., April 1980, AD A084-411
TR-234	Herlihy, Maurice P. TRansmitting Abstract Values in Messages, S.M. Thesis, EE & CS Dept., May 1980, AD A086-984
TR-235	Levin, Leonid A. A Concept of Independence with Applications in Various Fields of Mathematics, May 1980
TR-236	Lloyd, Errol L. Scheduling Task Systems with Resources, Ph.D. Dissertation, EE & CS Dept., May 1980
TR-237	Kapur, Deepak Towards a Theory for Abstract Data Types, Ph.D. Dissertation, EE & CS Dept., June 1980, AD A085-877
TR-238	Bloniarz, Peter A. The Complexity of Monotone Boolean Functions and an Algorithm for Finding Shortest Paths in a Graph, Ph.D. Dissertation, EE & CS Dept., June 1980
TR-239	Baker, Clark M. Artwork Analysis Tools for VLSI Circuits, S.M. & E.E. Thesis, EE & CS Dept., June 1980, AD A087-040

TR-240	Montz, Lynn B. Safety and Optimization Transformations for Data Flow Programs, S.M. Thesis, EE & CS Dept., July 1980
TR-241	Archer, Rowland F., Jr. Representation and Analysis of Real-Time Control Structures, S.M. Thesis, EE & CS Dept., August 1980, AD A089-828
TR-242	Loui, Michael C. Simulations Among Multidimensional Turing Machines, Ph.D. Dissertation, EE & CS Dept., August 1980
TR-243	Svobodova, Liba Management of Object Histories in the Swallow Repository, August 1980, AD A089-836
TR-244	Ruth, Gregory R. Data Driven Loops, August 1980
TR-245	Church, Kenneth W. On Memory Limitations in Natural Language Processing, S.M. Thesis, EE & CS Dept., September 1980
TR-246	Tiuryn, Jerzy A Survey of the Logic of Effective Definitions, October 1980
TR-247	Weihl, William E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables, S.B.& S.M.Thesis, EE & CS Dept., October 1980
TR-248	LaPaugh, Andrea S. Algorithms for Integrated Circuit Layout: An Analytic Approach, Ph.D.Dissertation, EE & CS Dept., November 1980
TR-249	Turkle, Sherry Computers and People: Personal Computation, December 1980
TR-250	Leung, Clement Kin Cho Fault Tolerance in Packet Communication Computer Architectures, Ph.D. Dissertation, EE & CS Dept., December 1980

TR-251	Swartout, William R. Producing Explanations and Justifications of Expert Consulting Programs, Ph.D. Dissertation, EE & CS Dept., January 1981
TR-252	Arens, Gail C. Recovery of the Swallow Repository, S.M. Thesis, EE & CS Dept., January 1981, AD A096-374
TR-253	Ilson, Richard An Integrated Approach to Formatted Document Production, S.M. Thesis, EE & CS Dept., February 1981
TR-254	Ruth, Gregory, Steve Alter and William Martin A Very High Level Language for Business Data Processing, March 1981
TR-255	Kent, Stephen T. Protecting Externally Supplied Software in Small Computers, Ph.D. Dissertation, EE & CS Dept., March 1981
TR-256	Faust, Gregory G. Semiautomatic Translation of COBOL into HIBOL, S.M. Thesis, EE & CS Dept., April 1981
TR-257	Cisari, C. Application of Data Flow Architecture to Computer Music Synthesis, S.B./S.M. Thesis, EE & CS Dept., February 1981
TR-258	Singh, N. A Design Methodology for Self-Timed Systems, S.M. Thesis, EE & CS Dept., February 1981
TR-259	Bryant, R.E. A Switch-Level Simulation Model for Integrated Logic Circuits, Ph.D. Dissertation, EE & CS Dept., March 1981
TR-260	Moss, E.B. Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Dissertation, EE & CS Dept., April 1981
TR-261	Martin, W.A., Church, K.W., Patil, R.S. Preliminary Analysis of a Breadth-First Parsing Algorithm: Theoretical and Experimental Results, EE & CS Dept., June 1981

TR-262	Todd, K.W. High Level Val Constructs in a Static Data Flow Machine, S.M. Thesis, EE & CS Dept., June 1981
TR-263	Street, R.S. Propositional Dynamic Logic of Looping and Converse, Ph.D. Dissertation, EE & CS Dept., May 1981
TR-264	Schiffenbauer, R.D. Interactive Debugging in a Distributed Computational Environment, S.M. Thesis, EE & CS Dept., August 1981
TR-265	Thomas, R.E. A Data Flow Architecture with Improved Asymptotic Performance, Ph.D. Dissertation, EE & CS Dept., April 1981
TR-266	Good, M. An Ease of Use Evaluation of an Integrated Editor and Formatter, S.M. Thesis, EE & CS Dept., August 1981
TR-267	Patil, R.S. Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis, Ph.D. Dissertation, EE & CS Dept., October 1981
TR-268	Guttag, J.V., Kapur, D., Musser, D.R. Derived Pairs. Overlap Closures, and Rewrite Dominoes: New Tools for Analyzing Term Rewriting Systems, EE & CS Dept., December 1981
TR-269	Kanellakis, P.C. The Complexity of Concurrency Control for Distributed Data Bases, Ph.D. Dissertation, EE & CS Dept., December 1981
TR-270	Singh, V. The Design of a Routing Service for Campus-Wide Internet Transport, S.M. Thesis, EE & CS Dept., January 1982
TR-271	Rutherford, C.J., Davies, B., Barnett, A.I., Desforges, J.F. A Computer System for Decision Analysis in Hodgkins Disease, EE & CS Dept., February 1982
TR-272	Smith, B.C. Reflection and Semantics in a Procedural Language, Ph.D. Dissertation, EE & CS Dept., January 1982

PUBLICATIONS

TR-273	Estrin, D.L. Data Communications via Cable Television Networks: Technical and Policy Considerations, S.M. Thesis, EE & CS Dept., May 1982
TR-274	Leighton, F.T. Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI, Ph.D. Dissertation, EE & CS Dept., August 1981
TR-275	Kunin, J.S. Analysis and Specification of Office Procedures, Ph.D. Dissertation, EE & CS Dept., February 1982
TR-276	Srivas, M.K. Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications, Ph.D. Dissertation, EE & CS Dept., June 1982
TR-277	Johnson, M.G. Efficient Modeling for Short Channel Mos Circuit Simulation, S.M. Thesis, EE & CS Dept., August 1982
TR-278	Rosenstein, L.S. Display Management in an Integrated Office, S.M. Thesis, EE & CS Dept., January 1982
TR-279	Anderson, T.L. The Design of a Multiprocessor Development System, S.M. Thesis, EE & CS Dept., September 1982
TR-280	Guang-Rong, G. An Implementation Scheme for Array Operations in Static Data Flow Computers, S.M. Thesis, EE & CS Dept., May 1982
TR-281	Lynch, N.A. Multilevel Atomicity - A New Correctness Criterion for Data Base Concurrency Control, EE & CS Dept., August 1982
TR-282	Fischer, M.J., Lynch, N.A., Paterson, M.S. Impossibility of Distributed Consensus with One Faulty Process, EE & CS Dept., September 1982

TR-283	Sherman, H.B. A Comparative Study of Computer-Aided Clinical Diagnosis, S.M. Thesis, EE & CS Dept., January 1981
TR-284	Cosmadakis, S.S. Translating Updates of Relational Data Base Views, S.M. Thesis, EE & CS Dept., February 1983
TR-285	Lynch, N.A. Concurrency Control for Resilient Nested Transactions, EE & CS Dept., February 1983
TR-286	Goree, J.A. Internal Consistency of a Distributed Transaction System with Orphan Detection, S.M. Thesis, EE & CS Dept., January 1983
TR-287	Bui, T.N. On Bisecting Random Graphs, S.M. Thesis, EE & CS Dept., March 1983
TR-288	Landau, S.E. On Computing Galois Groups and its Application to Solvability by Radicals, Ph.D. Dissertation, EE & CS Dept., March 1983
TR-289	Sirbu, M., Schoichet, S.R., Kunin, J.S., Hammer, M.M., Sutherland, J.B., Zarmer, C.L. Office Analysis: Methodology and Case Studies, EE & CS Dept., March 1983
TR-290	Sutherland, J.B. An Office Analysis and Diagnosis Methodology, S.M. Thesis, EE & CS Dept., March 1983
TR-291	Pinter, R.Y. The Impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits, Ph.D. Dissertation, EE & CS Dept., August 1982
TR-292	Dornbrook, M., Blank, M. The MDL Programming Language Primer, EE & CS Dept., June 1980
TR-293	Galley, S.W., Pfister, G. The MDL Programming Language, EE & CS Dept., May 1979

TR-294	Lebling, P.D. The MDL Programming Environment, EE & CS Dept., May 1980
TR-295	Pitman, K.M. The Revised Maclisp Manual, EE & CS Dept., June 1983
TR-296	Church, K.W. Phrase-Structure Parsing: A Method for Taking Advantage of Allophonic Constraints, Ph.D. Dissertation, EE & CS Dept., June 1983
TR-297	Mok, A.K. Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment, Ph.D. Dissertation, EE & CS Dept., June 1983
TR-298	Krugler, K. Video Games and Computer Aided Instruction, EE & CS Dept., June 1983
TR-299	Wing, J. A Two Tiered Approach to Specifying Programs, June 1983

Progress Reports

1. Project MAC Progress Report I, to July 1964, AD 465-088
2. Project MAC Progress Report II, July 1964-July 1965, AD 629-494
3. Project MAC Progress Report III, July 1965-July 1966, AD 648-346
4. Project Mac Progress Report IV, July 1966-July 1967, AD 681-342
5. Project MAC Progress Report V, July 1967-July 1968, AD 687-770
6. Project MAC Progress Report VI, July 1968-July 1969, AD 705-434
7. Project MAC Progress Report VII, July 1969-July 1970, AD 732-767
8. Project MAC Progress Report VIII, July 1970-July 1971, AD 735-148
9. Project MAC Progress Report IX, July 1971-July 1972, AD 756-689
10. Project MAC Progress Report X, July 1972-July 1973, AD 771-428
11. Project MAC Progress Report XI, July 1973-July 1974, AD A004-966
12. Laboratory for Computer Science Progress Report XII, July 1974-July 1975, AD A024-527
13. Laboratory for Computer Science Progress Report XIII, July 1975-July 1976, AD A061-246
14. Laboratory for Computer Science Progress Report XIV, July 1976-July 1977, AD A061-932
15. Laboratory for Computer Science Progress Report 15, July 1977-July 1978, AD A073-958

16. Laboratory for Computer Science Progress Report 16, July 1978-July 1979, AD A088-355

17. Laboratory for Computer Science Progress Report 17, July 1979-July 1980, AD A093-384

18. Laboratory for Computer Science Progress Report 18, July 1980-June 1981, A 127586

19. Laboratory for Computer Science Progress Report 19, July 1981-June 1982, A 143429

Copies of all reports with A, AD, or PB numbers listed in Publications may be secured from the National Technical Information Service, U.S. Department of Commerce, Reports Division, 5285 Port Royal Road, Springfield, Virginia 22161. Prices vary. The reference number must be supplied with the request.

OFFICIAL DISTRIBUTION LIST

1984

Director — 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research — 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 — 6 Copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center — 12 Copies
Cameron Station
Alexandria, VA 22314

National Science Foundation — 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 — 1 Copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

Dr. G. Hopper, USNR — 1 Copy
NAVDAC-OOH
Department of the Navy
Washington, DC 20374

ATE

LMED

-8