PROJECT MAC PROGRESS REPORT X, JULY 1972-JUNE 1973

E. Fredkin

Massachusetts Institute of Technology

Prepared for:

Office of Naval Research

December 1973

BIBLIOGRAPHIC DATA SHEET	1. Report No. Progress Report X	2.	3. Recipient's Accession No. AD 771 428
4. Title and Subtitle Project MAC Progress Report X			5. Report Date December, 1973 6.
7. Author(s) Project MAC participants-Prof. E. Fredkin, Director			8. Performing Organization Rept No. MAC-PR-10
9. Performing Organization Name and Address Massachusetts Institute of Technology Project MAC 545 Technology Square, Cambridge, Mass. 02138			10. Project/Task/Work Unit No. 11. Contract/Grant No. N00014-70-A-0362-0001 N00014-70-A-0362-0006
12. Sponsoring Organization Name and Address Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301			13. Type of Report & Period Covered Progress Rpt. 6/72-6/73 14.

15. Supplementary Notes

16. Abstracts

Final Summary Report of Progress made at Project MAC during the period 6/72-6/73.

AD 771 428

17. Key Words and Document Analysis. 17a. Descriptors

Real-Time Computers
On-Line Computers
Multi-Access Computers
Dynamic Modeling
Heterarchical Programming
Computer Systems
Artificial Intelligence
Computer Languages
Computer Networks
Information Systems
Programming Languages
Computation Structures
Automata Theory

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
This document has been approved for public release and sale; its distribution is unlimited.	UNCLASSIFIED	153pp.
	20. Security Class (This Page) UNCLASSIFIED	22. Price 4.75 / 1.45

PROJECT MAC PROGRESS REPORT X

ib.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

PREFACE

Project MAC was organized at the Massachusetts Institute of Technology in the spring of 1963 for the purpose of conducting research in the fields of Machine Aided Cognition and Multiple-Access Computer systems. This research has led to the development of many innovations in computer technology, among them the development of the Compatible Time-Sharing System (CTSS) and Multics.

During the year ending June 30, 1973, Project MAC was reorganized into four major divisions: Fundamental Studies, Computer Systems Research, Programming Technology, and Automatic Programming (including Mathlab and PLANNER). MAC has 294 people associated with it, including: 28 faculty members mostly from the Electrical Engineering and the Mathematics departments; 49 staff members (DSR Staff and support staff); 105 graduate students; 73 undergraduates and 7 guests.

The Fundamental Studies division consists of two subgroups. The Theory of Computation group has been involved in the investigation and characterization of various complex logical and linguistic problems in order to find more efficient methods of computation. The Computation Structures Group has been investigating the dynamics of interacting systems and the structure of the individual systems in order to place computer systems design on a more rational basis.

The Computer Systems Research division has been responsible for the development of the Multics System in conjunction with Honeywell, Inc. During the past year, Honeywell announced that Multics would be offered as a standard product using their 6180 computer system. This marks the completion of Project MAC's involvement with Multics, which began in 1964. The activities of the Computer Systems research division have therefore become concerned less with professional programming development and more with academically oriented research, especially in the field of protection of information within a multiple access computer system. In addition, research is being conducted on ARPA network protocols and modifications, and the development of models of the jobs presented to the Multics system and the system's response to them.

The Programming Technology Division has been mainly concerned with the development of the "Dynamic Modeling" computer system (D.M.S.) as a software laboratory for research in programming. The system is intended to facilitate the formulation of computer programs by the development of more efficient utilities, the use of on-line interaction between programmer and system, powerful high-level languages, graphic as well as alphanumeric display, and on-line library and documentation of procedures and data.

The Automatic Programming Division consists of three subgroups: the Automatic Programming group, Mathlab, and PLANNER. The Automatic Programming group has been concerned with development of a system which can represent knowledge about a known problem on a higher level of abstraction than has previously been possible. The models which have been used in developing this system include Management Data Processing and information

and decision systems. Its purpose is to manipulate domains where much is already known about a given problem, but where there are no means for considering sub-contexts within the domain.

Mathlab's largest contribution this year has been the implementation of the MACSYMA System with M.I.T. and around the country. Further work is being done to debug this system, in addition to which MACSYMA has been made operational within the Multics system.

Project MAC has been sponsoring the development of PLANNER in conjunction with the Artificial Intelligence Laboratory. PLANNER is a system concerned with the development of knowledge-based programming, that is, a program which has significant knowledge of its own structure and purposes. This system is based not on a hierarchical approach, but rather a modular one. These modules are called actors. The PLANNER group has also been working on the construction of a Programming Apprentice which will make it easier for expert programmers to do knowledge-based programming.

In addition to these four major areas of research, Project MAC participated in a scientific exchange program between the National Academy of Sciences of the U.S.A. and the Academy of Sciences of the U.S.S.R. Professor Victor Briabrin visited us from Moscow during the period of October 1972 to April 1973, during which time he studied various high-level programming languages (such as LISP, CONNIVER, and PLANNER) in order to compare them with similar programs being developed in the U.S.S.R.

During the past year the basic program of Project MAC was funded by the Advanced Research Projects Agency (ARPA) and, in the area of Fundamental Studies, by the National Science Foundation (NSF). Individual projects were supported by the Rome Air Development Center, and IBM.

ADMINISTRATION

Prof. E. Fredkin	Director
Prof. S. S. Patil	Assistant Director
D. C. Scanlon	Administrative Officer
G. B. Walker	Business Manager
A. D. Egendorf	Director of Information Services
B. H. Kohl	Librarian
C. P. Doyle	Administrative Assistant

Undergraduate Students

R. Elkin

Support Staff

S. A. Bankole-Wright
G. W. Brown
L. S. Cavallaro
R. B. Combs
J. A. Darcy
L. L. Gammell
D. Kontrimus
E. Y. Lewis
M. K. Martucci
E. T. Moore
D. S. Niver
E. M. Roderick
M. A. Stein

COMPUTER SYSTEMS RESEARCH

Academic Staff

Prof. F. J. Corbató

Prof. M. D. Schroeder

Prof. J. H. Saltzer

DSR Staff

C. C. Garman

R. K. Kanodia

R. F. Mabee

K. J. Martin

E. W. Meyer, Jr.

N. I. Morris

M. A. Padlipsky

K. T. Pogran

M. B. Weaver

D. M. Wells

Graduate Students

D. D. Clark

R. J. Feiertag

R. M. Frankston

B. S. Greenberg

S. M. Hansen

D. H. Hunt

S. Kuo

P. A. Janson

L. J. Scheffler

A. Sekino

J. A. Stern

V. L. Voydock

[redacted]

Undergraduate Students

R. G. Bratt

D. Bricklin

J. M. Broughton

M. G. Chang

T. L. Davenport

L. J. DeRoma

D. K. Gifford

P. A. Green

M. Gross

R. H. Gumpertz

G. Harris

P. A. Karger

R. S. Lamson

D. A. Moon

A. Nourse

J. A. Pineda

D. P. Reed

C. D. Tavares

J. B. Williams, Jr.

J. D. Williams

Support Staff

O. D. Carey

D. E. Cohen

C. P. Doyle

S. D. Grant

D. L. Jones

M. F. Webber

COMPUTER SYSTEMS RESEARCH

Guests

Prof. K. Ikeda

K. Oda

M. Miyazaki

COMPUTER SYSTEMS RESEARCH

A. INTRODUCTION

The most significant benchmark this year was the announcement by Honeywell Information Systems, Inc. that the Multics system, the object of a joint research and development project since 1964, would be offered as a standard product using their 6180 computer system. This announcement heralds completion not only of the Multics project itself, but also of the successful transfer of expertise and knowledge from Project MAC to Honeywell, so that both maintenance and development can continue in Honeywell's hands.

The year was also marked by a continuing, and now essentially complete, transition of the Computer Systems Research Division from a professional programming development team to an academically oriented research organization. Thus, the number of undergraduate and graduate students in the division has climbed from a low of two (in 1966) to 23, and the number of professional programmers has dropped from a high of about 28 (in 1967) to six. Correspondingly, the activities of the division have shifted to research topics which can take advantage of the unique laboratory environment represented by Multics.

These activities fall into four major categories of Computer Systems Research. The first category is measurement of statistical properties of the presented load on the M.I.T. production Multics site, and development of models of both the presented load and of the system's response to that load. Judging from the number of spontaneous inquiries, both the measurements themselves and the models are of great current interest to manufacturers who seem to be developing product lines with virtual memory and other sophisticated features. The second category of activities are those related to the ARPA network, both working with other network participants in developing protocols, and also in modification to the Multics/ARPANET interface to respond to new protocols and to better integrate the ARPANET as a standard facility of a computer utility. The third category of activity is advanced research on the protection of shareable information stored in a multiple-access computer utility. This topic has recently become one of high interest, with IBM and several other organizations rushing to obtain some useful results. Since our group has had a long-standing interest in the subject, it is tackling some relatively advanced problems in the area: better definition of the essential central security kernel of a general purpose system, and methods of certifying the correctness of an implementation of that kernel. The fourth and final category of activities are several joint projects with other groups which, as will be explained in detail later, support the general research goals of the division.

B. MEASUREMENT AND ANALYSIS OF COMPUTER SYSTEMS

Activities in measurement and analysis have been almost exclusively the province of students. The objective in this area is to learn how to predict the performance effect of a proposed system design. On the hypothesis that many future system designs will have functional properties similar to those of Multics, it is an especially interesting system to measure. The availability of a measureable system running with a real load has led to a burst of activity in this area, and the performance of a wide variety of measurements:

- A doctoral thesis developing a hierarchical model of the Multics multiprogramming and demand paging algorithms was completed by Akira Sekino. This thesis was significant for its ability to predict the actual performance of Multics under load, yet using mathematically tractable models. The thesis is available as Project MAC Technical Report TR-103.

- In last year's progress report, a linear model of paging behavior was reported. The model relates the number of memory references (the "headway") between missing pages to the size of the paging memory. For memory sizes below 4 million words, a simple, linear relation was observed. During this year, a paper was written and submitted describing the model, and further measurements have been made exploring the shape of the headway function in the region above 4 million words. These measurements indicate that the linear approximation describes the behavior of the M.I.T. Multics installation quite accurately for memory sizes up to 8 million words, but that an exponential approximation may be better above that point. These measurements are of considerable interest to system designers, who need information about the potential performance effect of the large primary memory systems which are becoming economically feasible with recent advances in Large Scale Integration (LSI) production technology. A Master's thesis by Bernard Greenberg will be available in a Project MAC Technical Report.

- A method of measuring a single user's load on primary memory in a paging environment, in order to estimate program "size" and also to provide a reasonable charge for usage was tried, evaluated, and then added to the standard Multics system. The principle of the method is as follows: for a given size of memory, a "large" program would be expected to cause more missing page faults than a "small" program. Thus, a simple page fault count could provide a crude estimator of program size. In a multiprogramming environment, however, the amount of memory available to a program may be different every time the program runs. Thus, a simple page fault count would provide a quite variable estimate. On the other hand, since the page fault count climbs when this memory is

smaller, and vice-versa, the product of memory size and number of page faults should be a relatively stable number, but one which is larger for larger programs. (To the extent that an individual program follows the linear paging model, the measure should be perfectly constant with different memory sizes.) The new charging scheme uses this general strategy, and produces a memory usage measure which seems to be proportional to program size, and which varies with a ten-fold change in memory size by no more than 30 to 50%. Currently, this scheme is documented in the form of three internal working papers, two by Robert Frankston, and one by Prof. J. Saltzer.

- In early 1972, a drum space allocation and access request scheduling strategy called "folding" was implemented on the Multics system. This algorithm traded effective drum storage capacity for drum access time by maintaining multiple identical copies of each page on the drum, spaced equally around the drum circumference. When a page on the drum is to be read, the copy closest to the drum read heads is used, thereby reducing the drum access time.

  The reduction of drum size due to folding causes a redistribution of secondary memory access requests between drum and disk (the third level of memory). An analytic model of the drum behavior under the folding strategy (a variation of a model due to Coffman) was constructed, and using the previously mentioned "linear model" for the paging behavior of the Multics system, an analysis was developed to predict the mean access time of the combined drum-disk memory system as a function of the number of drum folds. Experiments were conducted with several possible configurations of the Multics system under a benchmark load to verify the analysis, and to verify that the number of drum folds actually being used in normal Multics service is optimum. A paper describing these results, by Lee Scheffler, has been submitted to the Fourth ACM Conference on Operating Systems Principles, to be held in October, 1973.

- In the past, disk subsystem design, equipment and configuration selection, and interface algorithm decisions have usually been made informally, without hard data to compare the performances of alternative disk subsystem architectures. A class of infinite-population queueing network models are being developed for predicting the probability density function of disk subsystem access time, given specifications of disk subsystem equipment, configuration, and load of arriving access requests. The models are unique in that they are effective for the types of loads typically encountered in virtual memory systems which use the disk subsystem as a paging device. The models and analysis methods are applicable to a wide range of disk subsystem types, including both fixed- and movable-head disks, single or multiple channel disk subsystems, and non-pre-emptive priority arrangements for expediting the service of some access requests at the expense of others. Straightforward analytic methods are used to derive relationships between access time, configuration, and load, which are then solved numerically. A

set of programs have been developed which evaluate the models and can be used to experiment with proposed disk subsystem configurations. A Master's thesis describing this work, by Lee Scheffler, is in preparation.

- Many problems in computer system performance modelling and evaluation require the manipulation of probability density functions and the solution of complex queueing system problems. Analytic methods are of limited applicability to such problems because unrealistic assumptions often must be made in the name of mathematical tractibility. In the course of research on disk subsystem performance evaluation, a system of programs was developed on Multics for performing numerical computations on probability density functions. Several basic primitives are currently implemented for: creating probability density functions of the common analytic shapes (exponential, hyperexponential, Erlang, normal, uniform, impulse), or any combination of these, or of any shape specified by a table of sample points; for combining probability density functions (weighted summation, convolution); and for displaying computing statistics (mean, variance, percentile points), and computing derived probability functions (cumulative probability distribution functions). These basic primitives are combined with iterative techniques for the solution of simple G/G/n queueing systems (General arrival discipline/General service discipline/n independent identical servers), and for more complex queueing systems. It is expected that, as performace evaluation research continues, this system of programs prepared by Lee Scheffler will see use in the construction and solution of more accurate models of computer system performance.

- In systems with virtual memory, dynamic control of the level of multiprogramming is needed to maintain a balance between unusable idle time and time spent doing page retrieval, such that overall system throughput is optimal. For dynamic control, a simple method of estimating the size of each program is needed. A new estimating algorithm, based on extrapolation of the previously observed paging rate of a process, (using the linear paging model for extrapolation) was proposed and implemented, in a test version of Multics. Measurements are not yet complete, but the scheme has already proven to be at least as effective as the currently implemented, very complex, heuristic estimator. There is now growing evidence that the earlier, first attempt at an estimating algorithm treats large programs very poorly. An undergraduate thesis by David Reed has been completed on this subject and he is continuing to experiment with the technique.

In addition to the measurement and analysis activities mentioned above, comparison of performance of Multics on the Honeywell 645 computer with that on the newer 6180 computer is underway, but not yet complete. Initial results indicate that the hardware processor is about twice as fast, and that the replacement of the rotating drum with a bulk core has reduced multi-programming and therefore paging by enough to give an overall performance increase factor of three between the two systems. Also, informal measurements of the traffic flowing through the ARPA network attachment have been used to guide the activities of the network group, reported in the next section.

C. ARPA NETWORK ACTIVITIES

Because of the large amount of production programming which has marked the CSR division's activities in the network area in the past, most work in this area has been carried out by staff programmers rather than the students. This year, student participation is increasing. At the same time, the group is becoming more active in network development activities.

Two significant revisions of the Multics ARPA network software were accomplished during the year. The first of these was to revise an assumption that other hosts have relatively large buffering capabilities. This assumption, made incorrect by wide use of the Terminal Interface Processor, led to a design based on servicing the network once per interaction with a human user or his program at the other site. Widespread use of the TIP, with its small buffers, produced traffic with many network transactions for each message, putting a severe strain on the initial design. The revised design, which responds to small transactions on an interrupt basis, reduces both the real-time delays in using Multics from the network and also the overhead costs at the price of increased complexity in the central core of the supervisor. The revised design has been in operation since October, 1972, and has proven quite satisfactory.

The second major software revision was to convert from a half-duplex network interface to a full duplex one, separating reading and writing onto two hardware channels. This change was made after concluding that the half-duplex connection is not adequately supported by the network itself. (The network resolves certain overload conditions in a manner which drops links to half-duplex connections.)

The full duplex connection required a new hardware interface, which was developed as an undergraduate thesis by Richard Gumpertz, with construction help from John Williams, another undergraduate. This new interface was also designed to operate with either a local or a distant network interface port, and to operate with the Honeywell 6180 IOM rather than with the older Honeywell 645 GIOC thereby permitting these two other changes to be anticipated and accepted smoothly. Parts and engineering assistance were supplied by Honeywell, in return for which Honeywell will be permitted to use the design in other attachments of 6000-line computers to ARPA-like networks.

Related to conversion to the Honeywell 6180, which is located in a different building, a second Interface Message Processor (IMP) has been ordered for installation near the 6180. This second IMP will permit attachment to the ARPANET of the Honeywell 6180 "development" machine, the planned Project MAC terminal system, the Artificial Intelligence Laboratory "mini-robot", and the M.I.T. 370/165, all in addition to the present three PDP-10's and the 6180 Multics "Service" machine.

In the protocols area, members of the group were quite active in the evolution of the new File Transfer Protocol (used for moving files from one system to another) and the re-design of the Telnet Protocol (used for setting up Teletype-like

terminal connections) in conjunction with other members of the Network Working Group. The problem of arranging for file access while maintaining privacy was one primary issue which is still only partly resolved. In another area, Michael Padlipsky has proposed a "unified user-level protocol" which is intended to facilitate use of different operating systems by people who have not made themselves expert in the idiosyncrasies of those systems, by providing a universal interface to common functions.

On the implementation level, the major addition was a File Transfer Protocol server which responds to file transfer requests arriving from other sites. A File Transfer command, a new Telnet command for use from Multics in accessing other network sites, and an I/O system interface module which permits any Multics program to direct input or output to a network link, are all in experimental use in the user interface area. A first implementation of programs which merge the ARPA network mail facility with the Multics mail facility was completed. A facility to automatically detect the need for and perform typewriter case-mapping was added. This facility (in principle unneeded according to network protocol rules) allows use of Multics from sites which do not yet provide network standard upper/lower case facilities. The re-initialization logic of the Network Control Program has been improved, thus allowing it to automatically respond to and recover from a wide variety of error conditions. The Network driver program has been revised to be compatible with a standard interface to the system operator, and of course, numerous bug fixes were made along the way.

Use of Multics via the ARPA network has increased over the year. New metering and reporting software was implemented in the Fall, which shows that logins per month increased from 254 to 950 between September, 1972, and April, 1973. At the April level, network use accounts for about 10% of all logins at the M.I.T. Multics site. The metering software has also established that in the same period, the network indirect cost (that is, extra Multics overhead involved because the network connection was used) has dropped from about 100% to about 7%, largely because of the software changes reported above.

D. PROTECTION OF INFORMATION

In this category of activity are several long-standing interests as well as a substantial new activity. The long-standing interests relate to providing mechanisms in the Multics design which permit controlled sharing of information with security against unauthorized intrusions.

A doctor's thesis by Michael Schroeder was completed this year, describing a design by which general protected subsystems may be implemented using a domain scheme. A protected subsystem is a collection of programs and data with the property that the data may be accessed only by the programs of the subsystem, and that the programs may be entered only at designated entry points. The general domain model improves on the earlier ring model in that it does not constrain protected systems to be hierarchically arranged when more than one is used in a single computation.

Schroeder's thesis goes into details of both a processor architecture and also a file system design which support protected subsystems; both are relatively small (though intricate) departures from typical current-day system designs, and thus appear that they would be quite practical to implement. The thesis is available as Project MAC TR-104.

A second doctor's thesis in the area of protection, by Leo Rotenberg, is in progress. Rotenberg is exploring the consequences of attaching restrictions to information in such a way that even after it is released to a program, the restrictions continue to operate. He has also developed a very interesting method of controlling who may change access specifications in a computer system. Basically, he permits a hierarchical control, but with constraining protocols. With Rotenberg's scheme, for example, one could arrange that a person's manager could have access to his personal files, but only after obtaining the agreement of another, higher-level manager. This example is only one of many possibilities. This thesis will be available as a Project MAC Technical Report when it is completed.

In a related activity, Richard Bratt has completed a bachelor's thesis which involves devising system support software which allows easy construction of user-provided subsystems in the protection-ring environment of Multics. Although protection rings are provided by the hardware of the 6180, and two rings are used by the Multics supervisor to protect itself, user applications of protection rings have so far been limited to special cases, since the file system provides no way of cataloguing protected subsystems. Bratt's thesis is concerned with appropriate cataloguing and user interface facilities.

As Honeywell transferred Multics from the 645 to the 6180 computer, the software which simulated rings of protection was dropped out in favor of the 6180 hardware support. Although the processor time to switch rings has dropped dramatically (from 3 milliseconds down to 15 microseconds) the performance improvement so far achieved is modest, since with simulated ring software, ring crossings had been minimized in frequency. The primary gain remains to be realized, as redesign of the central core of the system can now be carried out without the need for minimizing ring-crossings, thereby leading to probable simplification of the central core.

A detailed paper summarizing the design of the Multics information protection system was written by Prof. J. Saltzer. This paper has been submitted to the Fourth ACM Conference on Operating Systems Principles to be held in Yorktown, N.Y., in October, 1973, and it will also be made part of the introduction of the Multics Programmer's Manual.

Work on a new activity has begun: the redesign of the central core of the Multics system (taking advantage of the hardware rings as well as new insight) to produce a potentially auditable version of the parts of the system which affect security. This new activity is a fairly ambitious one, probably requiring about three years, and work this year has been confined to studying various aspects of system organization which can

potentially be made more methodical, and therefore simpler, and thus smaller, as needed for auditability. Some of the areas currently being studied include:

Design (and propagation of the design through the central core of the system) of a more uniform approach to coordination of parallel processes. The basic strategy change is to allow several processes to operate in the same address space. This strategy change will allow, for example, the efficient handling of small network transactions on a scheduled basis rather than an interrupt basis. Many other activites which currently require elaborate coordination strategies (e.g., stopping a process when the user presses his attention key) can similarly be simplified if this change is made. Richard Feiertag is developing this topic.

Identification of the implementation consequences of using a single, system-wide address space with universal segment identifiers, in contrast with the present Multics scheme which uses a separate address space for each process. A system-wide address space would apparently eliminate large sections of the present supervisor which maintain maps of the individual address spaces; the purpose of this study is to understand just how much simplification could result. Although a revised hardware architecture would be necessary to exploit this simplification completely, even without revised hardware it may be possible to modularize and separate those parts of the system concerned with segment number mapping. Victor Voydock is developing this topic.

A doctor's thesis, by David Clark, is exploring a simple but complete I/O architecture which in hardware provides complete separation of independent users, so that the operating system need not include any of the usual, very complex, I/O strategy and interrupt facilities -- they may all operate in the protection environment of the user of the I/O device. This scheme, if implemented, would again provide a substantial reduction in the size and complexity of the central core of an operating system.

As can be seen, all three of the above activities are directed toward simplifying the system so as to make the remaining parts, which implement the security kernel, susceptible to methodical auditing. One final activity in this area has been an attempt to carry out an initial audit of the user/supervisor interface, to see both how many errors would be found and also to learn about how auditing can be made easier. The general topic of making a system auditable relates closely with several other research and development projects on certification of operating systems currently underway at other sites, and contacts have been set up with these other sites so that work may be coordinated.

E. MISCELLANEOUS ACTIVITIES

Several other activities have been carried out by CSR division members, sometimes in support of other groups with which joint projects are underway.

In a joint project with the Automatic Programming Division, and led by David Reed, a Multics LISP interpreter/compiler system which is completely compatible with the LISP system on the Project MAC PDP-10's was developed. The Multics LISP system was proven operational by the transfer of the Project MAC Symbolic Manipulator (MACSYMA), via the ARPANET, to Multics, followed by its complete and correct operation. A new LISP manual, describing the language now used on both the PDP-10's and Multics was written by David Moon and Alex Sunguroff.

Project MAC, in a joint venture with the M.I.T. Information Processing Center, has developed a specification for a large (8 million words) primary memory system to be attached to the M.I.T. 6180 (Multics) computer. Rapidly developing technology in Large Scale Integrated MOS circuitry makes such a memory economically practical, and research in Automatic Programming will soon require availability of such a large memory system.

The area of Multics documentation has been the last to be transferred to Honeywell. As a result, the Multics Programmers' Manual, through revision 14, was published by Project MAC, although future revisions are now expected to be handled by Honeywell. As part of revision 12, two new chapters of introduction to the console language and to the programming environment were written. The chapters provide a liberal collection of examples, and greatly ease the problem of a beginner trying to learn the system.

Coordination of planning for the Project MAC terminal system has been carried out with the help of Kenneth Pogran who has also coordinated the installation of data communication cables between the Project MAC building and the Information Processing Center building. The Electronics Systems Laboratory of M.I.T. has agreed to help develop a detailed implementation proposal and a prototype terminal.

COMPUTER SYSTEMS RESEARCH

PUBLICATIONS

1. Clark, D., "A Demonstration of the Multics System," Videotape recording, M.I.T. Center for Advanced Engineering Studies.

2. Multics Programmers' Manual, Part I, (Introduction), Revision 12, November 30, 1972.

3. Multics Programmers' Manual, Part II, (Reference Guide), Revision 14, April 30, 1973.

4. Multics Programmers' Manual, Part III, (Subsystem Writer's Guide), Revision 1, May 31, 1973.

5. Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," Project MAC Technical Report TR-104.

6. Sekino, Akira, "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems," Project MAC Technical Report TR-103.

PROGRAMMING TECHNOLOGY

Academic Staff

Prof. J. J. Donovan
Prof. J. C. R. Licklider
Prof. S. E. Madnick
Prof. N. P. Negroponte

DSR Staff

A. K. Bhushan
E. H. Black
M. F. Brescia
R. D. Bressler
M. S. Broos
A. L. Brown
M. A. Cohen
S. W. Galley
J. F. Haverty
P. D. Lebling
J. C. Michener
S. G. Morton
L. G. Pantalone
S. G. Peltan
C. L. Reeve
A. Vezza

Graduate Students

P. M. Allaman
S. E. Cutler
B. K. Daniels
J. D. DeTreville
G. J. Farrell
R. M. Fox
L. I. Goodman
P. W. Hughett
J. R. Johnson
D. Koenig
S. Kruger
W. J. Long
C. P. Mah
J. A. Melber
H. F. Okrent
M. S. Seriff
R. A. Stern
J. R. Taggart
R. W. Weissberg
P. Yelton

Undergraduate Students

H. R. Brodie
A. Y. Chan
C. C. Conklin
R. G. Curley
R. A. Guida
L. M. Gutentag
J. H. Harris
W. F. Hui
C. A. Kessel

Undergraduate Students (cont.)

G. D. McGath
J. A. Petolino
R. L. Prakken
L. M. Rubin
N. D. Ryan
C. A. Schweinhart
R. Swift
J. D. Sybalsky
M. E. Wolfe

Support Staff

A. J. Hicks
R. F. Hill
E. F. Nangle
S. B. Pitkin

PROGRAMMING TECHNOLOGY

A. INTRODUCTION

The main goals of the research and development program of the Programming Technology Division are computer facilitation of human programming and automatic programming. The approach to those goals involves interactive programming systems, computer graphics, and computer networks. Almost all the work to be reported has been done with a computer system called the "Dynamic Modeling System". Its hardware base is a Digital Equipment Corporation PDP-10 System computer with an Evans and Sutherland LDS-1 graphics subsystem. The operating system is ITS, developed by members of the M.I.T. Artificial Intelligence Laboratory. The computer, graphics subsystem, and operating system constitute the foundations for, rather than the focus of, the research and development program, and we refer to previous annual reports for descriptions of them. The work of the past year has been focused on:

1. Further development of the Dynamic Modeling System (DMS) as a software laboratory for research on programming.

2. Computer-aided programming.

3. Computer networks.

4. Automatic programming.

B. THE DYNAMIC MODELING SYSTEM AS A SOFTWARE LABORATORY

The Dynamic Modeling System was conceived of four years ago as a hardware-software system to facilitate the preparation, testing, modification, operation, and understanding of computer-program models. During the period of its development, the concept has broadened to include other kinds of programs than models, but the change in the purpose and nature of the system attributable to the broadening has not been great. The system is intended to facilitate one's movement -- and his understanding of his movement -- from an initial and perhaps nebulous conception of a problem to a sharp and definite formulation of a solution -- the formulation being a computer program, the execution of which solves the problem.

The basic idea of computer facilitation of programming, and of problem solving through programming, is a meld of several parts:

1. On-line interaction between programmer and system

2. Graphic as well as alphanumeric display

3. On-line library of tested procedures and data

4. High-level language with great expressive power

5. Efficient utilities

6. On-line documentation

7. Software capable of "understanding" software sufficiently well to contribute effectively to retrieval, testing, updating, and learning

An ideal programming and problem-solving system is envisioned as an integrated realization of those component ideas that presents its capabilities to users through a consistent, user-oriented language and that actively facilitates the users' mastery of its capabilities.

At the present time, all the listed components are well developed in the Dynamic Modeling System and contribute strongly to the effectiveness of the system as a medium for programming and problem solving. In our estimate, the system is among the best anywhere in respect of items 1, 2, 4, 5, and 7, is unique in terms of 3, and is well on the way to being unique in terms of 6. The Dynamic Modeling System is still weak, however, despite significant improvements during the past year, in terms of integration, consistency, and ease of mastery by people who are not experienced programmers. The shortcomings in those areas are due mainly to the fact that the Dynamic Modeling System was built on a base consisting of an operating system (ITS) and several basic utilities (such as the text editor TECO, the debugging aid DDT, and the assembler MIDAS) developed earlier within a value structure emphasizing power and sophistication as opposed to system coherence and ease of mastery. We have sought to construct a coherent superstructure on that base, but we have not entirely overcome a continuing dependence upon the rather diverse utilities.

The value of the Dynamic Modeling System as a software laboratory stems in large part from the LISP-like language MUDDLE, from the coherent programming system CALICO, and from the information retrieval system IRS. MUDDLE and CALICO have been greatly improved this past year, and a bridge between them, which has the effect of bringing all the capabilities of each within the reach of the other, has been brought into operation. The programs of IRS were complete at the end of the year, but IRS was operating with an incomplete data base. Nevertheless, it was evident that the long-awaited time was at hand when information about all the software in the system would be available through content-sensitive (as well as name-dependent) means, not only to users but to programs.

Underlying the development and use of the Dynamic Modeling System is a concept of programming in which there is a blending of the essentially "top-down" approach that seems to be basic to what is now called "structured programming" and of the "bottom-up" approach that is necessarily associated, at least to some degree, with use of a library of prepared procedure and/or data modules. In the DMS concept, the user-programmer should be able to put together an operable program or system of programs in a short time. He should then be able to test it, to explore and analyze its behavior and its

results, and to modify it -- to proceed through those phases iteratively or recursively -- with facility. The initial phase should be carried out with the aid of a high-level programming language and also with recourse to modules, already prepared and tested, that handle basic information-processing tasks in a reasonably efficient way -- with as much efficiency as is compatible with the moderate degree of generality of purpose that is essential to the success of a modular library. In later phases, when the general structure of the program or system has been defined and the "bottlenecks" have been located, the programmer may substitute special-purpose modules for some of the general purpose modules drawn from the library. If he does, and if they are not hopelessly ad hoc, he is expected, in the DMS concept, to document them carefully and submit them to the library for subsequent use.

An important part of our work the last two years (and currently) has been (and is) an implementation and test of the concept just outlined. Our experience to date convinces us that the implemented concept is sound and effective and that it is more realistic than programming concepts that tacitly assume that every programming task is begun with nothing to build on, from, or out of.

For purposes of explication, let us assume that programmer-user ABC wishes to begin work in the DMS in MUDDLE, but in such a way as to have available the resources of CALICO. The union of MUDDLE and CALICO is called "YDRA"* (abbreviated "Y"), and MUDDLE within YDRA is called "YM". ABC finds a free console and types "^Z LOG ABC↵ YM↵" -- where "^" means "hold down the CONTROL KEY while you press the key next indicated" and "↵" denotes a carriage return. He then works in MUDDLE in just the way he would work in an independent MU.MUDDLE (in the environment MU, to be explained) except that he can define MUDDLE functions that are actually indirect calls to CALICO subroutines -- of which there are more than 2,000, replete with on-line documentation, in the CALICO library.

To define the MUDDLE function INVERT.MATRIX to be the CALICO subroutine INVERT, the programmer simply types to YM:

```
<DEF.CAL INVERT.MATRIX (MA) INVERT>
```

Thereafter, he can use INVERT.MATRIX in his MUDDLE programming just as though he had defined it explicitly in MUDDLE. (Soon it will be unnecessary for the user to create the DEF.CAL link to CALICO. If there is not already a MUDDLE function by the same name, he will be able to refer to any CALICO function by its CALICO name or by a predefined synonym.)

YDRA can be entered as "YC" instead of "YM", whereupon the environment is CALICO backed up by MUDDLE instead of MUDDLE backed up by CALICO. At any time in YM the programmer can switch to YC by applying the MUDDLE function YC, and at any time in YC he can switch to YM by calling the CALICO

subroutine YM. But let us focus on YM for the present and give a brief description of MUDDLE.

1. MUDDLE

MUDDLE with the MU environment is simply MUDDLE with a set of MUDDLE functions and files that provide several basic conveniences, the most important of which is dynamic loading of MUDDLE functions and global data from a library. This library (MU.LB), which consists of a personal part and a public part, is physically but not conceptually distinct from the CALICO library. At present, MU.LB contains a few more than 1,000 functions. It is growing rapidly. With dynamic loading, the MUDDLE programmer or program can call or use any MU.LB function or global datum simply by referring -- inside angle brackets -- to its name or the name of a function that refers to it. Thus he has an immediately accessible vocabulary about as large as that of Basic English. Adding to this the DEF.CAL facility mentioned earlier, which gives him fairly ready access** to the more than 2,000 CALICO subroutines, one arrives at a "functional" vocabulary that begins to approach what we think it will take to turn 75 percent of a typical programming task into the specification of top-level flow control and the preparation of calling sequences.

This is not the place, of course, to offer a detailed presentation of MUDDLE. Let it suffice to give a very brief description of the language and the status of its implementation and refer to two documents:

MUDDLE was designed by LISP devotees (Sussman, Hewitt -- of the M.I.T. Artificial Intelligence Laboratory -- and Reeve -- of the Programming Technology Division of Project MAC) to provide a more readable syntax, more data types, readier extensibility, a better base for graphics and networking, and several other features desirable in the implementation of PLANNER-like languages. The implementation of MUDDLE has been, for more than a year, in the hands of Reeve, Daniels, and Brodie. During the past year, the evaluator, the input-output facilities, the interrupt-handling section, and the declaration section of the MUDDLE interpreter have been rewritten and greatly improved. In addition, considerable progress has been made in the development of a compiler (Reeve, Daniels, Pfister).

*Paying our respects to the HYDRA operating-system project at Carnegie-Mellon University, which may have a prior claim to the name, we are changing the name of our HYDRA to "YDRA". The allusion to multiple heads remains the same.

**It may be worthwhile to DEF.CAL the whole CALICO library into MU.LB. That would eliminate the definitional step and make the subroutines almost as immediately accessible to YM as the MU.LB function.

The present MUDDLE compiler handles all but two of the things that can legally arise, and (together with declarations, which are optional insofar as operability is concerned but essential to efficiency) it yields functions that run from 5 to a hundred times as fast as their interpreted counterparts. There is, fundamentally, an inverse relation between "expressive power" (which implies that many decisions are left to be made by the interpreter) and the "running speed" of a language/implementation. MUDDLE has great expressive power, and interpreted MUDDLE is quite slow. The game is to have or put or leave as much expressive power as possible in the language -- and then provide a declaration facility with which, after everything has been tested and proven, the programmer can specialize the program with respect to particular types of data and thereby make it possible for a compiler to generate efficient (though not as diversely applicable) code. The MUDDLE compiler effort is moving in that direction. It still has a long way to go.

MUDDLE is described in A MUDDLE Primer? by Pfister [32], and the built-in operators of MUDDLE are listed in A MUDDLE Micro-Manual by Daniels [5].

2. CALICO

CALICO* is an environment for the kind of programming in which the efficiency of the resulting code is a major consideration or in which it is advantageous to be in direct contact with a subsystem-oriented command interpreter or with a large library of tested subroutines. CALICO is almost consistent with MUDDLE in respect of basic (primitive) data types (Broos, Haverty, Lebling) [1, 6, 8, 9, 10, 14, 20] -- about as consistent as it is possible to be while both software systems are growing. In respect of programming language, however, CALICO is quite different from MUDDLE: whereas MUDDLE is a single-language system, CALICO is an environment within which several different languages may be employed.

The assembly language of CALICO is MIDAS reinforced by a set of conventions called "Convention II" [2, 12, 13, 15-26, 28, 29, 34-36, 38] and a collection of special MIDAS macros and subroutines. The conventions define and the special macros implement five types of subroutines together with corresponding declarations and calling-and-returning sequences [33, 37]. The types range from very simple and very fast to quite complex and sophisticated but correspondingly less lightning-like. Subroutines of all but the simplest type are dynamically loaded if they are not already in main memory when called. The conventions also define, and the special subroutines implement,

*The acronym "CALICO" is derived from "CAll-and-return mediator", LIbrary", and "COmmand interpreter".

the basic data types and their access methods. Thus, as used in CALICO, MIDAS verges on being a high-level language. Some of the people who use it regularly argue that it provides most of the advantages of a high-level language without imposing constraints in the sometimes essential area of efficient coding.

As a result of the diligent efforts of Okrent and Sybalsky, CALICO offers a limited version of PL/1. The PL/1 subset compiler permits (but of course does not require) the intermingling of PL/1 and assembly language statements. It is a pre-processor which produces MIDAS output, which is then assembled into machine code. The PL/1 procedures are CALICO subroutines and can call and be called by other CALICO subroutines.

At a still higher level in the scale of languages is Lebling and Haverty's CHILL -- "Calico's HIgh-Level Language". It is quite similar to MUDDLE in many ways but is somewhat simpler and more directly associated with the subroutines of the CALICO library. Indeed, the results of CHILL programming (just as of convention-governed MIDAS programming and CALICO PL/1 programming) are regular CALICO subroutines. In some task contexts, the choice between CHILL and MUDDLE is a matter of personal taste or ego involvement. In other task contexts, CHILL's advantage of rapid access to the CALICO library or MUDDLE's advantage of being more fully developed may dominate.

An important aspect of CALICO is its command interpreter (Seriff). It is of the class of command interpreters that accept and complete an incomplete command term if the characters thus far received match one and only one of the strings in the command set. CALICO's command interpreter is unusual, however, in that it can operate with compound command terms and provides completion within each component. Moreover, the command set is controllable by the programmer and his programs; the commands of various subsystems can be introduced into the command search path or removed from it dynamically, and new subsystems can be written with calls to the command interpreter as a regular CALICO subroutine. Indeed, this is one of the basic ideas of the DMS: that any software in the system should be callable as a function, subroutine, macro, ..., or data set from anywhere else in the system. Our earlier acknowledgement of weakness in respect of integration referred especially to the fact that that objective has not been reached in respect of the operating system and in several of the utilities not developed by the Programming Technology Division.

A call-and-return mediator comes into play whenever one of the top two classes of CALICO subroutines is called and again when it returns. The mediator serves several basic functions, such as saving and restoring the contents of accumulators, managing stacks, checking data types, and parsing calling sequences. In addition, in certain modes, it keeps histories of program execution to facilitate debugging, and it offers the user-programmer a chance to intervene at the

time of each call and each return. Inasmuch as a fuller account was given in the Annual Report for 1971-72, the foregoing may suffice for general description. The main advances during the past year have been in data typing, parsing of calling sequences, and stack management (Broos, Lebling, Harris, Haverty, Long).

Although the CALICO library treats every item it contains in the same way, the items range in fact from the lowest-level subroutines (that call no subroutines) to complete subsystems such as ARCHIVER, CHILL, DIRECT, DISKIO, GROWL, IRS, KDM, KDMX, NETWORK, POLYVISION and SDM. Every item in the library is documented under Convention II and is available on-line in both source-language and machine-language form. Since the collection of abstracts is now more than 6 inches thick, there is no way to give a detailed description of the library here. However, Table 1 lists some of the areas to which the subroutines apply and the approximate number of subroutines in each area. Table 2 lists the IRS categories and the distribution by number of subroutines and percentage of the 1412 subroutine entries currently in the data base. (Note that a subroutine may be a member of more than one category.)

TABLE 1. SUMMARY OF THE SUBSYSTEMS IN THE CALICO ENVIRONMENT

Name	Author	Description	Subroutine Entries
APLINE	Haverty	ASCII pipeline processor	25
ARCHIV	Haverty	File archiver	15
CALIB	Broos	CALICO library system (source files)	40
CHILL	Lebling	CALICO high-level language similar to MUDDLE	100
CHAREG	Hui Michener	Character recognizer	50
COIN	Seriff	CALICO command processor	50
CREF	Seriff	Cross-referenced listing of a file	15
DATA	Haverty Lebling Broos Long	Entire CALICO data-typing system, including location insensitizing, delayed release of data, and reading and printing	300
DEMONIT	Stern	YDRA debugging aid	35
DIRECT	Guida Broos	Personnel directory system	10
DISKIO	Haverty	Disk data paging, storage and retrieval	70
ESP	Galley	Event simulator and presentor, graphical and on-line debugging aid	25
FTP	Chan Bhushan	ARPANET file transfer program	25
GROWL	Michener	Graphical output writing language	50
INTERRUPT PACKAGE	Seriff Hughett	Routines for handling software interrupt	30
IPC	Haverty	Inter-process communication	10
IRS	Broos	Information retrieval system	25

Table 1 (continued). Summary of the Subsystems in the Calico Environment.

Name	Author	Description	Subroutine Entries
KERNEL	Reeve Seriff Michener Long Brodie	Kernel of the CALICO system	100
KDM	Haverty	Keyed data manager	10
KDMX	Lebling	Tree-structured disk storage system	25
LXTEXT	Haverty	Dictionary based text processing	100
NETWRK	Seriff Bhushan Chan	ARPANET user and server TELNET programs	100
POLYVISION	Michener	E&S picture display area manager	25
RJE	Guida	ARPANET remote job entry	15
SDM	Broos	String data manager	30
UTILITY	*	General utility routines	500

*Contributors too numerous to list.

TABLE 2. DISTRIBUTION BY CATEGORY OF THE 1412 SUBROUTINES CURRENTLY IN THE IRS DATA BASE. (NOTE: SUBROUTINES MAY EXIST IN MORE THAN ONE CATEGORY.)

CATEGORY	NUMBER OF SUBROUTINES	% OF TOTAL
DATA MANAGEMENT	172	14%
DATA SET HANDLING	129	11%
DISPLAY	203	16%
INPUT/OUTPUT	258	22%
INTERRUPT HANDLING	13	1%
STAT/MATH	67	4%
NETWORK	91	6%
STRING	172	14%
UTILITY	498	43%
NO CATEGORY	159	13%

---

3. IRS

One facet of the exploration of which the DMS is the focus is the question of programmers' "fluency". Can programmers master a large set of software modules in approximately the same way most people master a large vocabulary of words and idioms of natural language? In each domain, programming and natural language, the number of terms and modules needed may be somewhere in the range from 5,000 to 50,000. In each domain every term and every module is complex and stands in complex interrelation with many other terms and modules. It is evident from our experience that some programmers who participate in the development of a library of software modules can develop a high degree of fluency in the use of those modules. Given a program's name, such a programmer can remember and explain what it does and how it relates to data types and to other programs -- he can do that for perhaps 950 out of a 1000 MUDDLE functions and probably almost as well for CALICO subroutines. It is much more difficult to remember the name of the module given its function. Even with MUDDLE functions named according to a system of naming conventions, with which one of us has had about a year's intensive experience, in at least half of the

instances it takes some searching to find the exact name of a desired function.

The problem of finding modules in a software library that satisfy specified descriptions is formally very much the same as the problem of "intellectual access" to documents in a book-and-journal library. The solution, also, appears to be formally very much the same: an information retrieval system. (But of course the parameters of on-line software retrieval systems and conventional document retrieval systems may be quite different.) We have explored most of the conventional library techniques -- index cards (including edge-notched cards), computer-generated lists (including inverted lists and indexes) posted on the walls near the consoles, review articles, human librarians, a descriptor-based on-line retrieval system, and informal communication among "authors". All have proven valuable, but the only techniques that appear to meet the time press of on-line programming are on-line, interactive retrieval techniques.

To be on-line and interactive is necessary but not sufficient. The attributes of completeness, automated search, and fast response are essential. Two of the information systems we have tried were found wanting mainly because of gaps in the information base: an old ITS program called "INFO" and a new, not yet completed MUDDLE function called "?" (Licklider, McGath). The "?" subsystem of CALICO's command interpreter and two subsystems for examining CALICO's library documentation had the advantage of gap-free completeness but suffered because one had to know the name of what he wanted to learn about or else search by scanning. The descriptor-based retrieval subsystem that we built at the outset of the project was on-line and provided automatic search via file inversion, but it suffered from slow response because it operated in MULTICS and so, to use it, one had to switch to the DMS NETWORK program, log into MULTICS, and start up the retrieval subsystem. (It was not economical to stay logged in all the time just to retrieve information.) These experiences convinced us that an on-line retrieval system was precisely what we needed, but also that it must not lack in completeness, responsiveness, or search power.

IRS (Broos) now provides those* and other described features in one general-purpose CALICO subsystem, and preliminary tests suggest that it will indeed solve the problem of finding modules pertinent to specific needs that arise during programming. IRS is available to MUDDLE as well as to CALICO. The IRS programs are table-structured and table-driven and thus wholly independent of the content of the data base to which they are connected. They can handle many categories of information: programmer's name, module's name, argument data types and structures, result data types and structures, descriptors, Dewey-Decimal-like classification,

*Except completeness, which refers to content.

and so on. They provide for multi-inversion of the data files and for automatic updating. They respond sufficiently rapidly, even with a large data base, that users will not be frustrated by delay.

We are now going to convert INFO and the MUDDLE "?" function over to IRS and to augment the descriptor part of the CALICO library documentation. (Descriptors have been neglected because, heretofore, there was no good way to use them.) Techniques for describing MUDDLE functions are being developed in connection with automatic programming (vide infra), and preliminary results of that work are being incorporated into IRS. And, finally, a protocol for MUDDLE programming is being developed that will make it difficult to define a MUDDLE function without documenting it -- and will deliver appropriate data directly to IRS. Thus we hope, during the coming year, to reach a point at which all MUDDLE and CALICO routines and the key utility programs are carefully described within a retrieval system that will give both human and automatic programmers ready access to essential information.

4. Graphics

Progress has been made during the past year toward incorporation of graphic techniques into programmers' regular working procedures. Many of the basic software modules necessary for picture definition, graphic display management, and graphic input management are ready for application. An applications program, FIGS, a program that facilitates the creation (by sketching on a graphic input tablet) and editing of figures to be printed on a line printer, was completed. Another, a Tool for Interactive Graphic Emergency Room Simulation (TIGERS, described later), is nearing completion. And a third application program, STATS, a statistical analysis and display package, was updated to provide an efficient interface to ARDS and IMLAC display terminals and to the "mouse", the two dimensional input device available at these terminals. Progress has been made on techniques for using graphic displays to aid in the debugging of programs. Two such programs, a Graphical Debugging Tool, GDT (Hughett) [11], and Execution Simulator and Presenter, ESP (Galley) [7], are discussed in the section on Computer-Aided Programming.

Advances in the CALICO environment's graphic capabilities centered upon the subsystems GROWL: GRaphics Output Writing Language (Michener); CHAREG: a CHAracter REcoGnizer for hand-drawn characters (Hui, Michener); and Polyvision, a subsystem for managing display surface area and input tablet surface area (Michener). Also, display subroutine modules and data sets were made uniform, with respect to a user's view of them, with PDP-10 CALICO subroutine modules and data sets, thus providing a capability to dynamically load them. Special memory storage allocation subroutines were implemented to minimize the number of tied-down display pages required by a process.

GROWL provides a capability for defining pictures in the CALICO environment. It provides the CALICO programmer with an integrated means for picture definition which is independent of the physical medium for output. It provides the necessary modules to allow a graphics program to use as its output device the LDS-1 display, the ARDS display and/or the IMLAC display. In addition, GROWL is intended to serve as a framework for the implementation of both the "SERVER" and "USER" ends of the ARPA Network graphics protocol.

CHAREG provides a graphics programmer with a subsystem for training and recognizing hand-drawn characters, drawn on a graphic tablet surface using a stylus. It provides a facile means for using and creating character definition dictionaries and for inserting and deleting character definitions into the dictionaries in an operationally natural way. It was converted to run under the CALICO environment. In addition, it was reorganized to purify much of its code, make it more efficient in terms of its execution time, and make the subroutines within the CHAREG subsystem more readily available in the CALICO environment. CHAREG is highly modular, and has several (8) existing encapsulations, one of which is used by the Polyvision graphics subsystem.

The Polyvision graphics subsystem acts as a manager which coordinates the action of multiple substantive graphic subsystems in their use of the display and tablet input area. The environment provided by Polyvision allows amicable sharing of the LDS-1 display surface area by all the display subsystems that a user may have running. At the user's finger tips, by means of stylus input, is a mechanism for creating, updating and deleting tasks. Polyvision also provides for (again under stylus control) expanding or contracting a task's allocated display area (rectangular areas called viewports) and the management of which task (including Polyvision) is to receive the tablet input. Polyvision accepts input from both the user's console and tablet. Tablet input may be in the form of light buttons or hand-drawn characters that are to be recognized by CHAREG before being interpreted by Polyvision or one of the other subsystems.

By using Polyvision, the human frees himself from the typical mode of operation imposed by a console, i.e., the mode in which he interacts with one subsystem for a period of time and then turns to another. Polyvision enables him to interact rapidly with many subsystems switching among them merely by moving his hand to a different part of the tablet surface. Also, he can visually compare the displays produced by different subsystems, or refer to one task's display while interacting with another.

This past year saw the MUDDLE GRAPHICS facility (Daniels, Black) [4] become fully operational. It provides the primitives to define a PICTURE. A PICTURE is a MUDDLE object whose type is PICTURE. It has special distinguishing attributes. A PICTURE can be: displayed on a display device (LDS-1, ARDS, IMLAC), erased, hit by a stylus, a sub-part of a

PICTURE, saved in a file, retrieved and displayed again, and processed for plotting on a CALCOMP plotter.

What GROWL provides for the CALICO programmer, MUDDLE GRAPHICS provides for the MUDDLE programmer. Namely, it is a method of defining pictures and the basic primitives to display them on a device. A more ambitious advancement of MUDDLE GRAPHICS is the Display Algorithm Language Interpreter (DALI), an experimental version of which has been designed and is under development by Pfister (a member of the Engineering Robotics Group of Project MAC).

The intent of DALI is to provide a means of definition and creation of (dynamically) changing pictures. The purpose of DALI is to allow the structure and dynamics of a picture to be separate from the structure of its driving application program. In general, such separation enhances modularity and decreases the complexity of interactive graphics programs. DALI has, in addition, been found to be extremely hardware independent.

DALI differs from previous display-oriented languages in that it does not treat a picture as passive data, but rather as a structure of active objects called picture modules. Each picture module contains an interrupt-driven procedure (a "daemon") and associated (named) inputs and outputs. Picture modules communicate through heterarchical input/output links; daemons are run in response to changes in input values (contents), and may compute and propagate new changes via their module's outputs. Facilities exist for structuring changes in nested groups of modules to be performed in parallel or in sequence (Pfister).

A further graphical effort in MUDDLE was the implementation of a set of primitives to provide a facile means of making two-dimensional plots on IMLAC and ARDS display terminals of equations with more than one variable (Ryan). Basically, all the variables but one are treated as parameters; convenient means are provided for changing which variable is not a parameter, as well as changing parameter values. It also provides a convenient means for plotting data on ARDS and IMLAC displays.

5. Hardware and ITS System Developments

The DMS PDP-10 acquired a DM-10 Memory Map from Systems Concepts of San Francisco, California, and a much simpler companion map for the Evans and Sutherland LDS-1 display. An ITS capable of performing swapping operations between primary and secondary memory was obtained from the Artificial Intelligence Laboratory. It was modified (Brescia, Cutler, Cohen) to provide a graceful transition from non-swapping to swapping domain for all DMS user level software. Specifically, modifications were made to: the disk code (Cutler) -- in order to provide a more rational utilization of the DMS disks than could otherwise be achieved; the console handling code (Cohen, Brescia) -- to handle DMS programs and

consoles; the Network Control Program (Brescia) -- to provide an operational network interface; and the display handling section (Black) -- to set up the memory map, field page faults, and tie down up to 20 pages of memory for the displays in use. Finally, an effort to merge the AI, ML, DMS systems under one source has been undertaken by Stallman, Greenblatt, and Knight of the Artificial Intelligence Laboratory, Jarvis of the Mathlab Group, and Brescia and Cutler of the Programming Technology Division. An additional 32K of memory (Vezza) was interfaced to the DMS bringing its total core memory size to 256K*. An effort has been started to make the LDS-1 consoles more easily available to Programming Technology Division programmers (Black) and to provide better character drawing capability (Morton, Black).

We have instituted a procedure (Brescia, Cutler, Vezza) for backing up files on 9 track magnetic tape and have implemented programs to carry out the procedure (Cutler). We have also developed procedures (Brescia, Cutler, Vezza) and programs (Cutler) for a file housekeeping system (alias Grim File Reaper) that is run approximately once a month for backing up and deleting files which have not been referenced recently.

6. Applications Programs

A substantive application program, Computer-Aided Evaluation and Design of Feedback Systems, CAEDFS (Cutler) [3], was completed this past year Another, a Tool for Interactive Graphical Emergency Room Simulation, TIGERS (Weissberg), is nearing completion. These two projects provided some experimentation with the MUDDLE and the MUDDLE GRAPHICS facilities to test their applicability to the implementation of application programs of this type. The experiment proved very useful, as it resulted in constructive feedback which influenced the MUDDLE and MUDDLE GRAPHICS implementations.

CAEDFS is a set of computer routines written in MUDDLE that analyzes feedback control systems, designs compensation networks, and outputs graphs of their predicted performance. Criteria used in the analysis phase are: gain-margin, phase-margin, unity-gain frequency, DC desensitivity, mid-band desensitivity, minimum phase before cross-over, step response, and maximum peaking. CAEDFS routines have incorporated in them knowledge and decision processes about the applicability and design of six types of feedback control system compensation networks -- three series type and three minor loop feedback type. The user may specify or let CAEDFS decide which type of compensation to use in a given situation in order to achieve the system design goals. The design of a compensation network is carried out using criteria, where applicable, such as: crossover frequency,

*K = 1024.

phase margin, minimum phase shift, extra poles, and available feedback gain. The predicted system performance can be output in the form of Bode, Nyquist, and unit step response plots.

The measure on the applicability of MUDDLE to applications such as this is of course the ease with which such systems can be built and how well they operate. Let it suffice to say that CAEDFS was designed and implemented in less than four months by a graduate student (Cutler, in association with Dr. J. Roberge, Professor of Electrical Engineering) as part of his master's thesis work. Further, it analyzes feedback control systems and designs compensation networks using on the order of 1 to 20 CPU seconds.

TIGERS is in operation and is nearing completion. It was designed and implemented by Weissberg (of the Programming Technology Division of Project MAC) in conjunction with Dr. R. C. Larson, Professor of Electrical Engineering and Urban Studies, and Dr. R. P. Mogielnicki, M.D., of the Cambridge Hospital Department of Community Medicine.

TIGERS is a tool for designers and administrators of hospital emergency rooms. Through graphics and interaction with the designer, a flexible modeling environment for the analysis of hypothetical hospital emergency rooms is created. Emergency room events, easily understood by people who have little or no mathematical or computer-oriented expertise, are presented in animated graphical form.

TIGERS allows the user to manipulate emergency room resources such as: number of beds, nurses, doctors, x-ray stations, etc. Also, it allows him to manipulate statistical parameters of the model such as: average arrival rate, probability of a patient requiring an x-ray, mean time at an x-ray station, etc. It is implemented in such a way as to provide interaction with the model in a natural and convenient manner through the use of tablet input and displayed light buttons.

C. COMPUTER-AIDED PROGRAMMING

Probably the greatest single computer aid to programming is a responsive, interactive computer system with good programming languages, a good library, good documentation, and good retrieval facilities. The Dynamic Modeling System, as thus far described, is in our assessment such a system. On that foundation, we are building an integrated array of aids to facilitate the following aspects of programming.

1. Administration of programming projects.
2. Communication among the members of a programming project.
3. Design, preliminary programming, and testing and evaluation of preliminary programs.

4. Programming conveniences

5. Editing

6. Debugging

7. Keeping track of software modules and moving them from initial operation through a series of steps to residence in a public library

8. Naming (within a system of naming conventions)

9. Documentation

10. Evaluation

11. Understanding programs and modules prepared by others

12. Maintenance of modules, programs, and systems

13. Understanding the process of programming

An essential aspect of aid to programming is to have much of the programming task already done and to have the results readily available for application. The libraries described in the preceding section provide the basic mechanism for ready availability. To have prepared in the past software that turns out to be useful in the present or will turn out to be useful in the future requires, in addition, a working philosophy relating to software generality, and that philosophy must be regarded, also, as a programming aid.

1. Administration of Programming Projects

This area we have only recently begun to explore systematically. We have experimented with use of a file area, ADMIN, as an exchange medium for goal statements and progress reports. We have a personnel information subsystem (Guida). A few of us have explored the facilities provided by the Network Information Center at Stanford Research Institute. Our experiences with those items indicate that, to be fully effective, an administrative subsystem must be consonant with four fundamental guidelines:

1. Administrative communication must pass through the computer and must automatically create the records required in administrative control.

2. All substantive work must be done within the computer system and must automatically create the records required in administrative evaluation.

3. Administrative information must be organized

for retrieval by description rather than retrieval by name.

4. The availability of computer means must not be allowed to induce overadminstration.

We are now planning an administrative subsystem, to be based on the information retrieval system, IRS, that will be (insofar as possible in an early version) responsive to the four guidelines.

2. Communication Among the Members of a Programming Project

Although still largely face-to-face and informal, communication among programmers has been greatly facilitated by computerized mail, both local and via the ARPA Network, and by announcements in system and subsystem heralds. To date, our mail facilities have been limited to person-to-person and person-to-specified-group, and local and network mail have been separate and distinct. We have designed and are implementing a new, integrated mail service that brings all the desirable mail and announcement features we know of into one consistent framework (Haverty, Bhushan, Vezza, Hart) and, in addition, incorporates features of descriptor-based dissemination and retrieval schemes (Broos) and of "teleconferencing" systems (Haverty, Bhushan, Vezza, Hart).

3. Design, Preliminary Programming, and Testing and Evaluation of Preliminary Programs

In this area, we have explored the use of MUDDLE as a tool for design and preliminary programming of software later to be implemented in CALICO. In one project, for example, one of us prepared a MUDDLE program to express the general idea of what was desired. Another then expanded the idea in MUDDLE, explored several ways of handling key problems, and then rewrote the whole thing in CALICO to achieve the required operational efficiency. It was obvious that this procedure was much more effective than other procedures within our capability would have been. We are thinking in terms of more objective comparisons, but objective comparison in the programming field is fraught with difficulty. We are working to develop the idea of designing a program by first modeling it and then progressively turning the model into a full-fledged program.

In both MUDDLE and CALICO we have simple facilities for analyzing the temporal performance of programs. We plan to develop evaluation subsystems that will deal with memory and storage space as well as with time.

4. Programming Conveniences

In several experimental MUDDLE programming environments, we have incorporated conveniences (Farrell, Stern, Licklider, McGath) somewhat similar to some originated by Teitelman in the environment of BBN-LISP (now INTERLISP). These include

spelling correction, selective undoing of computation with a display of recent history of computation, attaching and checking of programmers' intent with respect to MUDDLE objects (by means of a debugging monitor interfaced to structure and string editors), functions for using and updating a random-access documentation library, and display of progress of computations at variable speed and detail.

It is so easy to define a dozen or so simple functions in MUDDLE that (insufficiently "structured") programmers tend to forget what they have defined before they get all the components connected together. A convenience that overcomes that problem is an interactive definer that keeps a record and periodically files the record as well as the defined functions. The interactive definer (Licklider) makes it easy to revise definitions as they are being formulated and permits the programmer to defer specification of the argument list until after he has completed the body of the function. The definer demands documentation information as soon as each new function has been defined.

Another MUDDLE "convenience" -- one that would not be required in a fully integrated software system -- is a set of functions that causes certain utilities to perform specialized chores for MUDDLE programs. For example, the function TE (Long) sends two files to TECO, one containing commands and the other an object, and TECO automatically performs the commands with respect to the object and sends the modified object back to MUDDLE, which automatically resumes processing. Such functions simulate integration by "papering over", but they do provide a great increment in convenience over independent utilities.

In CALICO, most of the programing conveniences are incorporated into the system of MIDAS macros, the command interpreter, the call-and-return mediator, and the high-level language CHILL. In addition to those, there are TECO macros to facilitate programming and documenting in adherence to Convention II (Michener).

5. Editing

In the DMS now are text and object editors suitable for all the available languages. TECO and IMEDIT remain the main text editors but a new text editor (Farrell) [31] is available in MUDDLE. There has been some progress toward the implementation of a text editor in CALICO (Broos). MEDDLE, the main object (i.e., structure-oriented) editor for MUDDLE has been greatly improved this past year (Pfister, Farrell), and it has been adapted more or less satisfactorily for use with CHILL (Lebling). In addition, two sets of editing primitives have been developed in MUDDLE (Pfister, Licklider), and it is now convenient to incorporate editing operations into application programs. Also in MUDDLE is a special editor for editing collections or families of MUDDLE functions (Licklider). It makes "global" changes automatically to all the functions and implements "local" changes function-by-

function in response to directions from the keyboard or a file.

6. Debugging

During the past year, considerable progress has been made in computer-aided debugging. There have been two main focuses: (1) basic system organization to facilitate the detection and elimination of bugs, and (2) specific debugging tools.

Under the heading of basic system organization fall several developments in MUDDLE and in CALICO. The declaration subsystem of MUDDLE now performs type checking during interpretation (Reeve) and the error subsystem reports type inconsistencies. Functions are available for examining the MUDDLE stack in ways pertinent to certain error reports. The multi-process feature of MUDDLE has been exploited to enable one process to "single-step" another (Farrell, Daniels). In CALICO, the type checking provided by CHILL (Lebling, Haverty) serves to catch errors in CALICO subroutines as well as in CHILL functions.

Under the heading of specific debugging tools, the past year has seen major advances in the subsystems called "ESP" (Galley) and "GDT" (Hughett) [11] and the completion of tools for analysis of module interconnectivity (Wolfe) [7, 27], cross references (Seriff), detection of bugs by invoking tests during execution (Stern), and detection of parenthesis mismatches (Daniels). ESP (Execution Simulator and Presenter) has been rewritten in CALICO and is now a wholly regular subsystem (Galley). GDT (Graphical Debugging Tool) has been endowed with the capability of recording execution history over 10,000 or so instruction cycles and developing "influence nets" within the recorded span (Hughett) [11]. An influence net shows, for a selected computational event, all the preceding events that could have influenced it and all the succeeding events that it could have influenced.

In both MUDDLE and CALICO, facilities have been developed for analyzing and presenting the interdependencies of modules in complex programs. A MUDDLE function makes summaries of MUDDLE functions, listing arguments, functions called, atoms with global values other than functions, atoms with local values, and atoms with no values (Licklider). Other functions are being prepared to check consistency within families of summarized functions. In CALICO, the library-maintenance and information-retrieval systems jointly analyze interdependencies and check consistency (Broos). In large program systems, such static debugging procedures appear to be essential because dynamic debugging may proceed for hours without encountering a bug that is very much present and potentially disruptive.

7. Keeping Track of Software Modules and Moving them from Initial Operation Through a Series of Steps to Residence in a Public Library

This aspect of programming is handled in CALICO by the integrated subsystem of library maintenance programs developed during the past year (Broos, Haverty, Lebling, Michener). The library subsystem provides an entry repository for new programs and takes each program through steps of testing, document checking, and admission to the public library. A complex array of pointers defines, at every moment, the current configuration of the library and its documentation. A subset of the latter is the ABSTRACT BOOK, which has been published in an edition of 50 copies so that each DMS programmer can have one at his elbow. The library-maintenance system automatically keeps track of new accessions and periodically prints updates for the ABSTRACT BOOK. The CALICO library-maintenance and information-retrieval systems are reasonably efficient and capable of dealing with data bases of significant size. The thought has occured to us that it might provide a good base for coping with the software problems of a sector of a computer network.

The provisions for keeping track of software in MUDDLE are experimental and not as efficiently developed as those of CALICO, but they connect more closely with the ongoing work of the individual programmer. The arrangements for following the programmers work are part of a MUDDLE programming environment called "MU" (Gray, Licklider, McGath, Yap).

MU provides the programmer with a special set of functions and files (Licklider, McGath) and a dynamic loader (Pfister) that facilitate his work. The part of this apparatus that deals with keeping track of software includes the interactive function definer mentioned earlier. The defined functions and their documentation go to separate "step-1" files as soon as definition is completed. There are three higher levels of files called step-2, step-3 and step-4. Each level gives an indication of the state of the exactness of the function's documentation and the vigor with which the function has been tested.

There is a protocol for promoting functions from one level to another. In the final step in the move, functions and data sets (far more of the former than of the latter) that promise to be generally useful are moved into the MU environment's public library (step-4 files), and at that time the documentation pertaining to them is reviewed and, if necessary, a final update is made.

Each year about a dozen undergraduate students carry out programming projects in a Project Laboratory associated with the DMS. This year we hope to "debug" the software control system just discussed by using it in the Project Laboratory.

8. Naming

Systematic procedures for naming the objects of a complex software system may be almost as important as systematic procedures for describing objects -- or may not be; the question is open. We have experimented with several conventions for naming files (Martin) [29] and functions (Licklider). None has achieved unanimous acceptance, even within our compact group, probably because naming offspring is viewed as an individual or family right and an area reserved for idiosyncratic expression. Nevertheless, it is obvious that sharing of software resources would be fostered by some nonzero degree of agreement about how to name objects, and we are still exploring the matter.

A basic problem in software nomination is whether a name should reflect meaning in a substantive application area (e.g., COST.OF.BEEF) capable of providing strong semantic support to a person who is trying to understand a program, or whether the name should suggest the data type or some other syntactic aspect that would still have significance if the same software object were employed in a different application area (e.g., INTEGER.1 or DOLLARS.AND.CENTS.2). Our strong interest in the sharing of resources and in the software library concept biases us in the direction of syntax-oriented naming. Most of the names of atoms in the libraries of the MU environment are syntactic names such as "S" ("Structure"), "•LV" ("List of Vectors"), and "•V/V4I.V4R" ("Vector consisting of two Vectors, the first consisting of 4 Integers and the second of 4 Real numbers"). (This scheme is elucidated a bit further in the section on Automatic Programming.) In some instances, it is advantageous to incorporate some amount of substantive meaning into a name, to make the name somewhat semantic as well as syntactic, but in the scheme being discussed, the semantic component is always restricted to the field of information processing and not allowed to extend to truly substantive application areas. For example, "•AV/VNI•INDEX.•VNI•DOCUMENT.NAME" is a possible name for a vector of two vectors of equal length, the first consisting of integers representing indexes and the second of strings representing document names. However, in the notation scheme used in MU, there is a long list of abbreviations, and the example would actually be "•AV/VNT•IX.•VNT•DM.NA". Once mentioned in a function, the vector thus identified could be referred to as just "A", which is in a sense the name of its name. MUDDLE itself does not automatically recognize name equivalences of that kind, but it of course permits two or more atoms to identify the same object, and concurrent use of full and abbreviated names is readily achieved in special MUDDLE environments.

It is easy, as the foregoing examples suggest, to contrive a scheme so complicated that no one but the contriver will ever learn it. We are exploring several possible solutions to that problem. We have functions that analyze compound names into their components and routines that recognize compound names when a sufficient number of their

components are specified in any order. The completion scheme used in the CALICO command interpreter is of course applicable. In MU there is a computerized flash-card learning aid that makes the mastery of abbreviation conventions rather painless, even enjoyable. And there are name-translation functions that replace abbreviated name components by their expansions or vice versa. Indeed, for simple names, there is a print mode that displays such names in full even though they are represented in memory as abbreviations -- and a print mode that does the converse.

With the facilities mentioned, we are beginning to explore the feasibility of a system in which all the frequently used software concepts have convention-governed names and computer-processible definitions. The names will serve as declarations, and both programmers and programs will "know" or be able to figure out what they can legally do to and what can legally be done by any named object. This of course is just one approach. Another is to rely on separate declarations and let objects be named willy-nilly. The essential issue is not how to represent the distinctions; it is that the distinctions be made explicitly and consistently in a computer-comprehensible way and that they be preserved throughout a whole system of software.

9. Documentation

In the early stages of the project, members of the group viewed the preparation of documentation with great disdain. It was much more enjoyable and reinforcing to write and debug programs than to document them because documentation did not play an essential role in the system building until the system grew beyond the scope of unaided memory and informal communication. Sometime in the second year, documentation aids began to be developed, the painfulness of adhering to documentation conventions diminished, and the CALICO library began to grow and almost automatically to achieve a reasonable degree of consistency. The documentation aids constructed at that time are mainly TECO macros. They will perhaps eventually be replaced by an integrated CALICO documentation subsystem, but the macros are sufficiently effective that all that was urgently needed was organization and documentation of them. That was accomplished this last year (Michener).

In MUDDLE, the situation is now about like it was in CALICO two years ago except that, in MUDDLE, documentation is inhibited by the fact that it is usually easier to write and debug a function than to document it. It may turn out to be true that documentation does in fact have a smaller role to play in MUDDLE than in CALICO, but there is no question that documentation (especially computer-comprehensible documentation) is central to the concepts of software laboratory, software production facility, and automatic programming. We have therefore been exploring computer-aided documentation in MUDDLE. The exploration relates to the definer function and the naming conventions mentioned earlier and to a discussion in the section on Automatic Programming of

how to specify what functions do. Inasmuch as the exploration is in an early phase, those mentions may suffice for the present.

10. Evaluation (Assessment)

Although computer aids to evaluation of software is an important topic, we have just recently started to come to grips with one small part of it. A basic measure of the value of a software object is the number of times it is used. The library-maintenance system provides for each subroutine a list of all the subroutines that call that subroutine, and the system derives from the set of such lists a list of pairs that shows for each subroutine of the library how many callers it has (Broos). That list of pairs is itself a fairly good evaluation of the "callees". In addition, we have explored the feasibility of tallying every call to every mediated subroutine (Vezza), and we have a preliminary plan for a timing subsystem that will automatically compare various versions of the same MUDDLE function (Licklider).

11. Understanding Programs and Modules Prepared by Others

Documentation, conventions, and the inherent clarity of programming languages are of course basic to this topic, but the topic is open to study and facilitation in its own right. We have made just one beginning toward such study: an experiment in which several programmers were asked to debug and report on their debugging of a few briefly described programs in which bugs were planted (Schweinhart). There appear to be very great differences, even among programmers who spend several hours a day at the console, in understanding such programs well enough to debug them in any other way than by rewriting them de novo. For some, evidently, it is much easier to devise an algorithm than to figure out why a slightly wrong algorithm doesn't work. Perhaps the algorithm does not have to be wrong at all: it may be that some people are simply very much better at software synthesis than at software analysis.

ESP and GDT, mentioned earlier as debugging tools, are also effective aids in understanding other people's programs. The user-intervention feature of CALICO'S call-and-return mediator is also an effective aid: it permits one to examine the progress of a computation in steps corresponding to call-to-call, call-to-return, return-to-call, and return-to-return intervals, and so to step through a more complex program than can be examined instruction-by-instruction. In MUDDLE, we now have two functions (Farrell, Petolino) that display evaluation step by step. Brevity demands that an example be unrealistically simple:

```
Input:  <SET X <+ 1 <* 2 3>>>@
Output: <SET X <+ 1 <* 2 3>>>
        <SET X <+ 1 6>>
        <SET X 7>
        7
```

12. Maintenance of Modules, Programs, and Systems

This area is the focus of the CALICO library-maintenance programs (Broos) already mentioned several times. In MUDDLE, the pertinent work concerns testing MUDDLE functions and recycling defective library functions through the stepwise procedure described earlier (Licklider, McGath). The main goal in this area is a system in which there is an explicit representation of all the software ramifications and interactions. In such a system, a program will be able to know that changing module X will or may upset modules Y and Z, and in precisely what way -- and it will in some cases even be able to make compensating adjustments to Y and Z (and then determine the ramifications of those adjustments). This is a broad and deep subject, but valuable practical results have already been obtained by simply automating the bookkeeping of which modules refer to which other modules (Broos, Licklider).

It is widely understood from experience that complex software systems need to be exercised continually. Frequent exercise seems to keep the bugs out, or at least reveal bugs, motivate their elimination, and then reveal the bugs introduced by the elimination. We have explored the idea of systematically exercising software -- executing it under the control of a program that knows what should happen and checks that it does happen (Gray, Licklider). Systematic exercise has revealed flaws immediately after library updates, when the updating was still fresh in mind, that would otherwise have gone undetected long enough to become difficult to understand. We have some ideas about an exerciser that will not merely conduct "canned" tests but also devise new tests on the basis of documentation.

13. Understanding the Process of Programming

It is very clear that many programmers can program well but that few if any can explain how they do it. To develop a basis for automatic programming it is necessary to find out. To that end we have made a beginning on a MUDDLE subsystem that will create an annotated record of what programmers do (Licklider).

In this subsystem as it now operates, a record is made of every object the MUDDLE interpreter reads, the time at which the interpreter finshes reading it, every object returned by the interpreter after evaluating what was read, and the time at which the interpreter finishes the returning. To this record are added, in their entirety (even if not so read or returned) the definitions and documentation items provided through the definer and also notes prepared from time to time by the programmer to explain his motives and intentions. The notes and documentation are crucial to interpretation of the records. Every 40 console interactions, therefore, the subsystem checks to see whether or not the programmer has submitted the required number/amount of notes/documentation. If he has, he is allowed to continue, but, if he has not, the

subsystem asks him to remedy the lack and cycles him back through the request loop until he does so.

With the aid of his notes and documentation, a programmer can recall rather well what he did, and apparently also to a considerable extent why he did it, during a recent session at the console. It is difficult for another person to figure out exactly what happened, and more difficult for another person to get a good idea why, but it seems likely that the scheme can be developed into a facility for studying the process of programming.

We are quite aware, incidentally, that there are serious social implications in computer monitoring of human behavior. We do not propose to have the system monitor anyone's programming without his consent and his cooperation in interpreting the records.

D. COMPUTER NETWORKS

The work of the Programming Technology Division in the area of computer networks is focused on the ARPA Network. The work has shifted during the past year from development of software serving basic network functions to development and exploitation of the ARPANET as a virtual extension of the Dynamic Modeling System, as a pool of resources for use in computer-aided programming, and as a communication medium. Our main objectives in networking are to facilitate use of certain remote resources to such a degree that they appear to be integral parts of the DMS, to make a few of the subsystems of the DMS very conveniently available to remote users, and to advance the arts of person-to-person and program-to-program communication. Our interests are evolving in the direction of a software laboratory distributed among several network hosts yet functionally integrated and coherent.

Our Network effort has centered upon the development of: (1) effective network communication; (2) an experimental Virtual File Management System; (3) a network interface for MUDDLE; (4) programs for the automatic filing of data, collected by our SURVEY program, at the Datacomputer (DC) and the retrieval of SURVEY data; (5) a NETWORK MUDDLE; (6) programs in CALICO and MUDDLE to facilitate use of the ARPANET; (7) measurement functions to evaluate the performance of DMS network programs; and (8) the ICCC Special Project Demonstration.

Before we proceed to a discussion of these main network efforts, three system related tasks that were completed must be reported. At the end of the reporting period, an experimental Server TELNET (Chan, Brescia, Bhushan) behaving in accordance with the new TELNET protocol was operational on DMS socket 69 (decimal). The NCP of the new model ITS (swapping system) and our HOST-IMP hardware interface were mutually modified (Brescia) to make them compatible, and additional NCP calls were installed (Brescia) so that our sans-swapping ITS network software became operational with a

minimum amount of modification.

1. Network Communication

We are well into the implementation of a unified communication facility (Haverty, Bhushan, Brescia, Vezza) for both intra- and inter-system use. We are providing a number of facilitation functions, such as distribution by group name, and deferred distribution to remote ARPANET hosts. One important function will be to thread together communiques whose content is pertinent to a particular subject matter and provide an information retrieval system to facilitate extraction of information.

2. Virtual File Management System

A version of a Virtual File Management System (VFMS) (35) was designed, simulated, and implemented (Seriff). The goal of the system is to provide a mechanism whereby file directory operations and file access operations are specified in a uniform manner for all file systems on the ARPANET hosts that possess a VFMS Server File Transfer Program (FTP) that behaves according to the VFMS File Transfer Protocol. (Currently the implementation is operational in the three ITS environments at M.I.T.) Further, the design provides for a virtual file directory system that allows the names of files from any number of hosts to co-exist in a single directory. The VFMS maps ITS commands into host specific commands. The file directory structure of the VFMS is patterned after the MULTICS hierarchical file directory structure.

Currently, VFMS provides a capability to list a user directory, print a file on one's console, copy a file from one directory to another, append a file to a file, rename a file, create a link, delete a file, create a split file (a virtual file composed of constituent files that retain their identity) and create and maintain multiple copies of a file on different machines.

3. MUDDLE Network Interface

In keeping with the DMS philosophy of integrating programming functions, such as editing, debugging, and networking, into the programming environments of CALICO and MUDDLE, we have recently added network primitives to the MUDDLE environment (Reeve, Ryan). (Network primitives and a USER TELNET existed in CALICO prior to the beginning of this reporting period.) Addition of the primitives provided a basis for a USER MULTI-TELNET and USER MULTI-FTP facilities (Scandora, Bhushan). Also based on these primitives were three efforts described in greater detail in the next three sections: the SURRET subsystem, the NETWORK MUDDLE subsystem, and MUDDLE facilitation functions for the use of subsystems at remote hosts.

The integration of networking functions into the major subsystem programming environments is leading to the

implementation of programming conveniences for using remote resources on the ARPANET. These conveniences provide the user with a uniform view of many diverse systems and resources, and the presentation of that view is made in a manner that is consistent with what the user already knows about the DMS. The MUDDLE TELNET and FTP facilities presently provide a convenient mechanism for accessing resources on the ARPANET without the need or bother of leaving the MUDDLE environment. Also, many of the programming conveniences that exist in the CALICO environment are appearing in MUDDLE.

4. SURVEY

The SURVEY program collects data about ARPANET host status at 20-minute intervals. A host's status may be: host down, NCP not responding, initial connection aborted by foreign host, logger not responding, logger available, or undetermined. In addition, response time for a "request for connection" is collected for all hosts with status "logger available".

In conjunction with the SURVEY program, we have completed the development of:

1. A service for sending SURVEY data.

2. Storage of SURVEY data at the Datacomputer without human intervention.

3. A MUDDLE interface to retrieve from the Datacomputer ARPANET host status data collected by the SURVEY program.

A special ARPANET socket, socket 15 (decimal) at DMS, transmits the most recently collected SURVEY data each time a connection to the socket is established (Bhushan, Seriff). The data are formatted for use by an automaton. Connections are closed by DMS immediately after the data are transmitted. We plan to provide a similar service with a format suitable for human reading.

A cooperative project with Computer Corporation of America to store and retrieve SURVEY data with the aid of the Datacomputer was undertaken and completed this past year. The purposes of the project were: to determine whether the anticipated mass data storage of the Datacomputer could be made to appear to be an integral part of the DMS; to obtain some experience and competence with the Datacomputer and the Datalanguage; and to let us act as guinea pigs for and friendly critics of the developing Datacomputer facility.

The project entailed the implementation of programs that transmit SURVEY data to the Datacomputer automatically (Bengelloun, Bhushan) and a set of MUDDLE functions that provide a facility for specifying retrieval commands with MUDDLE syntax and semantics. The programs that transmit the SURVEY data to the Datacomputer attempt to do so after each measurement. If the transmission is unsuccessful, the data

are stored locally, and an attempt to transmit the accumulated data is made at the next measurement time. The MUDDLE subsystem SURRLT provides a facility for automatic connection to a special socket at the Datacomputer, the generation of Datalanguage from retrieval commands written in MUDDLE syntax and semantics, the transmission of a request for retrieval in Datalanguge syntax and semantics, the arrangement and display of the retrieved data in a form appropriate for human viewing, and the arrangement of the retrieved data in a form appropriate for processing by MUDDLE functions.

5. NETWORK MUDDLE

It is quite clear that in a heterogeneous network environment such as the ARPA NETWORK the modus operandi currently employed by local users is extremely cumbersome. This is so because the diversity of host operating systems would force a network user to employ many different operating procedures in order to use, or even to explore, the full range of resources offered by the network. It is not the employment of diverse operating procedures that in and of itself causes the main difficulty; it is having to learn diverse operating procedures. If a user identifies a resource that exists at a remote host on the network and that could help solve his problem, it impedes his progress if he must stop and learn the idiosyncrasies of the operating system in which the identified resource resides. His progress would be further impeded by the need to learn the idiosyncrasies of editors, file transfer mechanisms and other ancillary subsystems associated with the remote operating system. Additionally, file storage allocation must be obtained and file system conventions learned. Instead of having to learn a lot of new procedures, a user would like to get on with the business at hand, that is, the application of the identified resource to his problem.

It is not difficult, we have discovered, to make some resources available directly to the ARPANET. (By directly, we mean that a user who has general programming capabilities elsewhere need only concern himself with acquiring knowledge about the program he wishes to use and not about our operating system or ancillary programs.) Thus, the user is provided with the semblance or beginning of a network operating system environment. We have identified two DMS resources, MUDDLE and the DMS IMLAC assembler MIDASI, that we believe are potentially useful to the network community. A first pass implementation of a NETWORK MUDDLE (Bengelloun, Bhushan, Vezza) is already operational and a means of providing a NETWORK MIDASI is being planned (Brescia, Vezza).

The NETWORK MUDDLE facility will provide direct access to MUDDLE from the network. It will be usable by both humans and programs that exist at remote hosts. In this latter endeavor, we are type coding all of MUDDLE's responses (Bengelloun).

The NETWORK MUDDLE is operational on DMS socket 73 (decimal) and provides a user with an environment in which he can perform almost all the operations that a local DMS user

can. The differences between the NETWORK MUDDLE environment and the normal MUDDLE environment include: the NETWORK MUDDLE does not allow either output to DMS secondary storage or input from secondary storage other than from the NETMUD directory, while the normal MUDDLE allows both of these operations.

These secondary storage restrictions at DMS do not hamper the user because NETWORK MUDDLE provides a capability to NFLOAD files from and to NPFILE objects to remote hosts, and other similar facilities are being added. The NFLOAD and NPFILE commands accept arguments specifying source and target pathnames and access privileges at remote hosts. Figure 1 is a diagram illustrating the logical connections that may exist for a user accessing NETWORK MUDDLE from a TIP. As indicated, more than one file transfer connection is allowed. This provides a capability for files to have in them references to other files that are to be inserted in the input stream at the point of reference. The recursion depth is limited by the maximum number of simplex connections (16) permissible in NETWORK MUDDLE. (This is really a current ITS restriction on the number of software channels an ITS job may possess.) The input file to NETWORK MUDDLE can exist on any host computer on the ARPANET provided the host has a file transfer program that behaves in accordance with the standard file transfer protocol.

The best method of illustrating NETWORK MUDDLE is by example. In the example presented below the "remote host" is the DMS itself. We connected the CALICO USER TELNET via the network to the DMS NETWORK MUDDLE socket. We did this because it was convenient and because we wished to script the example (a facility existing in our CALICO USER TELNET, which files all terminal input and output). In the example we NFLOAD and NPFILE to a TENEX system at SRI-ARC. We begin by commanding our user TELNET to connect to socket 73 at DMS. The user's console input is underlined. Comments we have inserted are enclosed in quotes and prefixed by a semicolon. All lines beginning with a three digit integer are the remote host's server FTP responses. (In normal mode, these responses would be suppressed and the console output would not be cluttered up with them.) Everything else is NETWORK MUDDLE output.

@CONNECtion to host DM socket 73
completed.
MIT Dynamod System PDP-10
MUDDLE LISTENING-AT-LEVEL 1 PROCESS 1

<NFLOAD "MUDEX.TXT;1" "SRI-ARC" ("MIT-DMCG" "password" "3")>@
300 SRI-ARC FTP Server 1.27.0.0 - at THU 9-AUG-73 10:22-PDT
330 User name accepted. Password, please.
230 Password OK. Send ACCOUNT before writing any files.
200 Account command accepted.
200 Socket command accepted.
255 SOCK 3276932615
250 ASCII retrieve of <MIT-DMCG>MUDEX.TXT;1 started.
252 Transfer completed.
"DONE"

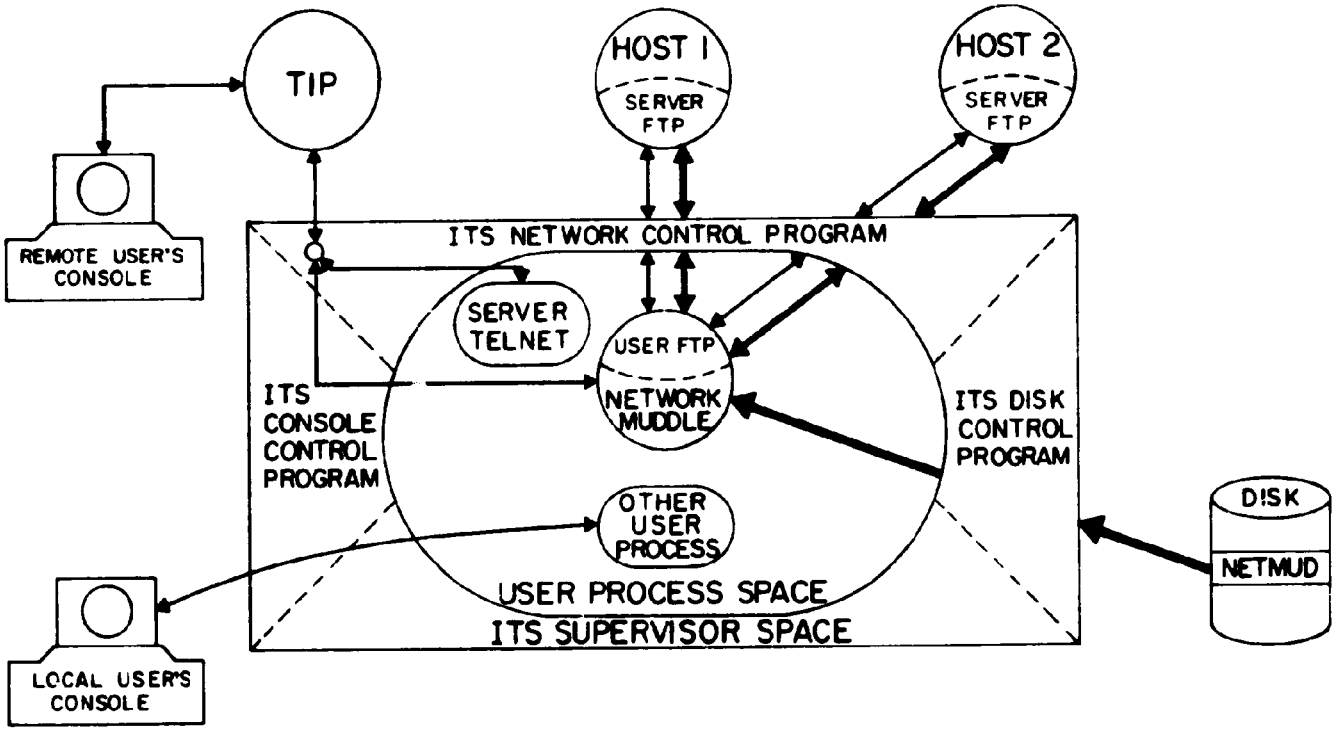


FIG. 1 NETWORK MUDDLE IN THE DMS

```
;" WE JUST LOADED FROM THE NIC A FILE
WHICH HAS A WHO? COMMAND; LET'S TEST IT"
<WHO>@

TTY UNAME  JNAME  CORE  TOTAL IDX
T00 MSB.SM DUMP   008   010   13
T05 SYS    SYS    052   119   32
             ...
             ...
             ...

DSN MSB    IRSUPD 090   098   10
FREE CORE  018 OUT 126ΛL"DONE"

;"OK NOW WE WANT TO CHANGE THE COMMAND NAME TO LISTF AND
SEND IT BACK TO THE NIC"

<SET MUDOBJ (SETG LISTF ,WHO?)>@
(SETG LISTF
      #FUNCTION (("AUX" CH)
           <SET CH <OPEN "READ" "TTY:.FILE. (DIR)">>
           <REPEAT () <PRINC <READCHR .CH '<RETURN>>>>
           <CLOSE .CH>
           "DONE"))
<NPFILE "MUDEX.TXT;2" <CHTYPE .MUDOBJ FORM> "SRI-ARC">@
200 Socket command accepted.
255 SOCK 3276932614
250 Store of <MIT-DMCG>
MUDEX.TXT;2;P777752;A3, ASCII type, started.
252 Transfer completed.
"DONE"

;"NOW LET'S RETRIEVE IT"
<NFLOAD "MUDEX.TXT;2" "SRI-ARC">@
200 Socket command accepted.
255 SOCK 3276932615
250 ASCII retrieve of <MIT-DMCG>MUDEX.TXT;2 started.
252 Transfer completed.
"DONE"
<LISTF>@

TTY UNAME  JNAME  CORE  TOTAL IDX
T00 MSB.SM DUMP   008   010   13
T05 SYS    SYS    052   121   32
             ...
             ...
             ...

DSN MSB    IRSUPD 090   098   10
FREE CORE  012 OUT 117ΛL"DONE"
;"IN <UNSOAK> MODE; THE FTP REPLIES AND COMMANDS ARE HIDDEN.
IF AN ERROR RESULTS IN FTP, MUDDLE'S ERROR FUNCTION IS
CALLED WITH THE FTP DIAGNOSTIC."
```

NETWORK MUDDLE is in a state of flux; it is changing rapidly because it is still a new development. We are implementing new commands to make the syntax and semantics of

NETWORK MUDDLE uniform with that of MUDDLE. Because NETWORK MUDDLE is changing so rapidly, the above example will in all probability no longer work by the time this report is printed. However, if anyone wishes to try an example, after connecting to the NETWORK MUDDLE socket type

<FLOAD "EXAMPLE">●

to obtain a set of instructions about an up-to-date example that will work.

A discussion on the important topic of whether or not such a service is practical given current network bandwidths is deferred to the section on Measurement of Performance of DMS Network Programs.

It has occurred to us that there may be something to be gained by merging the VFMS and NETWORK MUDDLE. It would provide a uniform user interface to all files on the ARPANET accessible to NETWORK MUDDLE. However, it would be yet another set of conventions to learn and at present little would be gained because most users know the conventions required by the host computers where their files exist -- and that is what NETWORK MUDDLE requires as a pathname argument. In the future, as users become less concerned about where files actually exist, as files migrate to where storage exists, and as mass storage becomes available on the ARPANET, it may be an attractive merger.

6. Facilitation Functions for Use of the ARPANET

A number of functions that facilitate the use of the ARPANET for a DMS user have been implemented in MUDDLE (Bhushan, Holman, Hart, Scandora) and CALICO (Bhushan, Bressler, Chan). These are intended to make the use of the network, by a DMS user, as painless and convenient as possible. Some examples of such facilitation functions are: the "who" command in CALICO, which takes as an argument a host name and returns a list of the users currently logged into that host; and the "NLS" command in CALICO, which allows one to type "NLS" and a password, at the appropriate time, and after a short wait one finds himself connected to NLS at SRI-ARC. There are a number of such facilitation features that perform most of the login ritual on behalf of the user, whenever he types a host name, so he doesn't have to remember the rituals for each computer he may use. Some of them are print, rename, copy, mail, delete, and listf. In addition to performing the login function, appropriate TELNET modes are set, i.e., character or line at a time, full or half duplex, etc. (In the new protocol, the TELNET modes will be negotiated.)

An InterEntity Communications protocol has been developed and programs behaving in accordance with the protocol have been designed and implemented (Bressler, Chan). These programs allow network users to link and communicate with each other. Also, we are presently experimenting with a system

that allows NLS journal submission of ITS created files (Bhushan).

We have experimented with some functions in MUDDLE that go beyond simple NETWORK facilitation. These functions provide invocation and use of programs at remote hosts in such a manner that they appear to reside on the DMS, that is to say, the programs are called without explicit intervention by a user. We have implemented the MUDDLE functions DIFF (Bhushan), INTEG (Holman, Bhushan, Vezza), and some other ancillary functions that call MACSYMA (MAC's SYmbolic MAnipulation system) to perform symbolic differentiation or integration. We have looked into providing calls to the CONSISTENT SYSTEM on MULTICS (Dehn, Vezza) and have some preliminary plans to do so.

The automatic call of programs in remote hosts is illustrated with the aid of MACSYMA (comments are spaced over and preceded by a semicolon):

```
SCENARIO FOR USING THE DIFFERENTIATE FUNCTION (VIA MACSYMA)

MUDDLE 42 IN OPERATION
<FLOAD "AKB;NAUTNE">@                    ; FLOAD from file.

"DONE"

<AMACSYM 1>@      ; This gets a MACSYMA at either MIT-MATHLAB or
                    MIT-AI and the argument "1" tells program
                    to supress remote computer's responses.

PLEASE BE PATIENT, MACSYMA LOADING MAY TAKE TIME
MACSYMA AT MIT-MATHLAB
T                          ; true response from program, false if
                             both AI and MATHLAB are down

<DIFF "XA3+4*XA2+7*X">@    ; to differentiate an expression in
                             string form

"3*XA2+8*X+7"              ; The program returns answer in string
                             form; computation is done by MACSYMA
                             at MATHLAB.

<DIFF "XA4+7*XA3+XA2+5*X+1" "X" "2">@    ; to differentiate an
                                           expression two times
                                           with respect to X

"12*XA2+42*x+2
<DIS>@                     ; to disconnect from MACSYMA

"CONNECTIONS CLOSED NOW"
```

The implementation of the DIFF and INTEG functions has made us acutely aware of one interesting fact. Basically, subsystems are written to interface to humans and their interfaces are not particularly suited for automata. To illustrate what we mean, let us suppose that our DMS program

has passed MACSYMA a function to integrate and is expecting to receive the integrated function back. But, before MACSYMA finishes the integration, the DMS program receives from the remote host, "System going down in 9:59", or "Fatal error", or any one of several less nocuous statements. Of course there is also the hoped-for possibility that the answer will be forthcoming; we must be able to recognize it also. How can programs analyze and take correct action on such replies? (The system-going-down message has interesting side effects -- suppose the system is to go down in about an hour and the task to be run takes well over an hour, on the average, to complete; clearly running the task is quite likely to prove futile.) Quite clearly, a program capable of handling all possible free format responses is beyond our current capability. A protocol specifying response types would in fact make message type recognition much easier. A simple set of types would be: expected system lifetime, fatal error, error in server process input stream, informative information -- something the user process can store for later perusal by a human, etc.

In conjunction with our NETWORK MUDDLE effort, we are currently building a taxonomy of NETWORK MUDDLE responses. The responses are being type coded. We hope to be able to collect them into categories that will enable an automaton to determine whether it is prudent to continue, or cause an interrupt to a higher level at the local host, or take other desired actions.

7. Measurement of Performance of DMS Network Programs

We have made some measurements on the performance of several of our network programs in order to obtain an understanding of the breadth of possibilities in using the ARPANET and also to discover whether our programs are grossly lacking in terms of performance. The measurement effort has proved quite fruitful: in a number of instances minor if not trivial modifications to network programs have increased their performance with respect to some measure by an order of magnitude or more. A measurement facility associated with the FTP (Bhushan, Chan) was implemented and since February has intermittently monitored file transfers to and from the DMS. A summary of the results is shown in Table 3. The measurements were taken on 265 file transfers during the period February 1973 to July 1973. In all 53.9 million bits were transferred. The DMS was a sender or receiver in both user and server mode, and both image and ASCII files were transferred. "Data recorded" distinguishes between whether the DMS functioned as a sender or receiver, user or server, and whether image or ASCII files were transferred.

From the data in Table 3 it is clear that we are not yet operating near the network bandwidth. The raw data show that bandwidths of better than 27 kilobits per second have been achieved, and that large files are transferred at significantly higher transfer rates than smaller ones. The observations collected thus far indicate that data transfer rates depend strongly on the input buffer size.

TABLE 3. MEASUREMENT OF THE DMS FILE TRANSFER PROGRAMS

Transfer mode	# Data Bits Transferred Megabits	Data Transfer Rate Bits per Second	CPU-time Seconds/ Megabit
Server Sending Data	3.8	7504	2
Server Receiving Data	19.8	7441	1
User Sending Data	8.7	7982	2
User Receiving Data	21.5	8839	5
ASCII 8 mode transfer	3.3	2435	12
36 bit image mode transfer	50.5	9472	2
Combined Data transfers	53.9	8042	3

An important question is: given the data transfer rates achievable on the ARPANET, are services like the MUDDLE NETWORK practical? We think that some such services are in fact practical, mainly because many programs perform computation on the input data stream which has the effect of drastically limiting the data transfer rate between secondary and primary memory. Taking the example of a MUDDLE FLOAD, the file is evaluated by the interpreter as part of the FLOAD process. Some simple measurements (Vezza, Bengelloun) made on the MUDDLE FLOAD function indicate that one can expect (currently), for typical MUDDLE text files, effective data transfer rates (fully evaluated) between primary and secondary memory of about 3-10 kilobits/second. (The 3 kilobits/second is more typical than the 10 kilobits/second.) This data rate can be and is achieved by data transfers on the ARPANET.

A RESTORE or a SAVE in MUDDLE (which restores or saves a MUDDLE environment) achieves a much higher data transfer rate than a FLOAD or PFILE. Under typical load conditions, one can expect effective data transfer rates between secondary and primary memory of 36 kilobits/second. This is close enough to the current maximum achievable network bandwidth that restoring or saving environments in NETWORK MUDDLE will be slower than in MUDDLE. However, this is done infrequently -- typically at start up and a couple of times during a console session, and restoring even large files such as the MUDDLE compiler is not likely to take more than 2 or 3 minutes.

8. ICCC

A highlight of the early part of the reporting period was the Special Project Demonstration of the International Computer and Communications Conference (ICCC) held in

Washington, D.C. Several members of the Programming Technology Division staff (Vezza, Bhushan, Bressler, Haverty, Lebling) helped organize and prepare scenarios for the Special Project Demonstration. Eight members of the Division attended the Conference (Vezza, Bhushan, Bressler, Haverty, Lebling, Licklider, Reeve, Seriff) to help set up the equipment and demonstrate resources on the ARPANET. Three others (Brescia, Chan, Cutler) provided technical backup to insure the availability of the DMS.

E. AUTOMATIC PROGRAMMING

The term "automatic programming" is being used these days both (a) as a general "chapter heading" to cover a wide range of approaches to improvement of the preparation of software and (b) as a specific identifier for (the development of) programs that write programs with little or no help from human beings. The earlier section on "Computer-Aided Programming" falls within the general sense of "automatic programming", but in this section the more specific sense is intended. In the Programming Technology Division, the level of effort contributing to this area is small, and the work itself less advanced than that of the Automatic Programming Division.

1. Automatic Composition of Functions from Modules

A fundamental and appealing idea in automatic programming is to provide an automatic programmer with a problem specification and a library of program modules and to have the automatic programmer select the required modules and compose a program that will solve the specified problem. An advanced version of such an automatic programmer would involve a problem-acquisition subsystem, and its modules would include generators of code as well as "canned" macros, functions, and subroutines. We are thinking about such matters and exploring some of the problems involved, but most of our tangible results thus far pertain to a quite simple system, an automatic composer of MUDDLE functions dealing with a very restricted domain.

LAP (Little Automatic Programmer) receives its problem specification in the form of a sample input-output pair such as

```
IN->(A B C D E F G H I J K L)

OUT->((A C E G I K)(B D F H J L))
```

That pair is intended to challenge LAP to find or compose a function that will put the odd-numbered components of the value of its argument into one list and the even-numbered components into another list and then return a list consisting of the two lists in sequence. (There are other possible interpretations of the sample pair, of course, but almost everyone sees the intended interpretation first, and so should an automatic programmer.) The symbols employed by the "client" in setting up the problem specification (the sample

pair) are not restricted to letters. They could be "JOE", "BILL", "PETE", ... or "217", "327", "182906", ... They serve only as names.

The module library of LAP consists of MUDDLE functions. Because LAP runs slowly and is used only as an exploring tool, not as a serious programmer, the library is usually restricted to about a dozen functions, but in principle there is no limit to its size.

The operation of LAP is illustrated by the script reproduced below. For this run, the library included 15 modules, three of which were:

REVERSE	reverses the order of the components of a list
ROTATE.RG	rotates the components of a list one place to the right
ROTATE.LF	rotates the components of a list one place to the left

The sample input-output pair was

```
IN->(A B C D E F G H I J K L)

OUT->((B D F H J L) (C E G I K A))
```

That pair was intended to convey to LAP a request to find or compose a function that rotates the components of the input list one place to the left and then subdivides the list into two parts, one containing the odd-numbered members and the other the even-numbered members. The version of the LAP in this illustration is 6. It explains what it is doing as it runs.

Lap.6 Script

As I understand the problem, I am to find or synthesize a MUDDLE function that will convert inputs like the sample input into outputs like the sample output. The sample input and output are:

```
<SET SMPL.IN '(A B C D E F G H I J K L)>

<SET SMPL.OUT '((B D F H J L) (C E G I K A))>
```

I shall try to find or synthesize a function that will meet the requirement. As I work, I shall display some of my intermediate steps. At certain points, I shall have a lot of processing to do, and at those points I shall fall silent for what will probably seem to be quite long intervals. My first step is to describe the sample input-output pair with the aid of descriptors. The reason for doing so is that I want to be able to select a small, pertinent part of my data base in which to search for a function that will effect the required

pair) are not restricted to letters. They could be "JOE", "BILL", "PETE", ... or "217", "327", "182906", ... They serve only as names.

The module library of LAP consists of MUDDLE functions. Because LAP runs slowly and is used only as an exploring tool, not as a serious programmer, the library is usually restricted to about a dozen functions, but in principle there is no limit to its size.

The operation of LAP is illustrated by the script reproduced below. For this run, the library included 15 modules, three of which were:

REVERSE	reverses the order of the components of a list
ROTATE.RG	rotates the components of a list one place to the right
ROTATE.LF	rotates the components of a list one place to the left

The sample input-output pair was

```
IN->(A B C D E F G H I J K L)

OUT->((B D F H J L) (C E G I K A))
```

That pair was intended to convey to LAP a request to find or compose a function that rotates the components of the input list one place to the left and then subdivides the list into two parts, one containing the odd-numbered members and the other the even-numbered members. The version of the LAP in this illustration is 6. It explains what it is doing as it runs.

Lap.6 Script

As I understand the problem, I am to find or synthesize a MUDDLE function that will convert inputs like the sample input into outputs like the sample output. The sample input and output are:

```
<SET SMPL.IN '(A B C D E F G H I J K L)>

<SET SMPL.OUT '((B D F H J L) (C E G I K A))>
```

I shall try to find or synthesize a function that will meet the requirement. As I work, I shall display some of my intermediate steps. At certain points, I shall have a lot of processing to do, and at those points I shall fall silent for what will probably seem to be quite long intervals. My first step is to describe the sample input-output pair with the aid of descriptors. The reason for doing so is that I want to be able to select a small, pertinent part of my data base in which to search for a function that will effect the required

transformation or for component functions out of which to synthesize such a function. My second step is to translate the sample input-output pair into a canonical form that is independent of the particular symbols used in the sample pair. This step takes me quite some time. When I have completed it, I shall display the canonic input and output.

```
<SET INPUT '[1 2 3 4 5 6 7 8 9 10 11 12]>

<SET OUTPUT '((2 4 6 8 10 12) (3 5 7 9 11 1))>
```

My third step is to substitute for the canonic forms of the sample input and output another form that is similar to an outline. In this form, there is a header, which I shall not explain here, and a body. The body consists of a list of positions and a list of the canonic symbols that occupy those positions. As soon as I have made the outlines, or 'inlines', as I call them (because the elements of the component lists are 'in line' in their lists rather than being paired with corresponding elements as in an ordinary outline), I shall display them.

```
<SET IN.ILN
     '((100 1)
       ((1 2 3 4 5 6 7 8 9 10 11 12)
        (1 2 3 4 5 6 7 8 9 10 11 12)))>

<SET OUT.ILN
     '((100 2)
       ((101 102 103 104 105 106 201 202 203 204 205 206)
        (2 4 6 8 10 12 3 5 7 9 11 1)))>
```

That concludes my preamble. Now I shall go into the main business of finding or synthesizing the required function.

```
ENTERING AP.6. FROM LEVEL 0

<SET LEVEL '1>
```

Now I shall look in my data base for a function, already prepared, that will meet the requirement. I shall place on a pushdown list the names of the functions I examine. The pushdown list is called 'PDL'. I shall prime it with the symbol 'ZZ', but that symbol will almost immediately be removed in favor of the first trial function name.

```
<SET PDL '(ZZ)>
```

Next, I shall find the output inline that would be yielded by each trial function. That output I shall call 'OUT.ILN.' (with a period on the end). I shall compare OUT.ILN. with the desired output 'OUT.ILN' inline, and, if they match, I shall conclude that I have discovered a ready-made function that will meet the requirement. If no trial output matches the desired output, I shall move on to try to synthesize a function out of available component functions.

```
<SET I '1>

<SET PDL '(IS.MOD.2)>

<SET OUT.ILN.
     '((100 2)
       ((101 102 103 104 105 106 107 108 109 110 111 112)
        (1 2 3 4 5 6 7 8 9 10 11 12)))>

<SET OUT.ILN
     '((100 2)
       ((101 102 103 104 105 106 201 202 203 204 205 206)
        (2 4 6 8 10 12 3 5 7 9 11 1)))>
```

... Several unsuccessful attempts omitted here ...

DID NOT FIND FUNCTION THAT FILLS THE BILL SO WILL
TRY TO SYNTHESIZE REQUIRED FUNCTION FROM COMPONENT FUNCTIONS

My approach, now, will be to select tentative 'first' functions and then, for each first function, to place 'second' functions in tandem, one at a time, to create a new, compound function that may meet the requirement. I shall test each tandem pair made with first function I = 1 and, if none is satisfactory, try another first function I = 2 and go through the list of possible second-functions again. The first function will be the inner function and the second function will be the outer function in a tandem such as:

```
        <DEFINE ALPHA (S) <SECOND.FUNCTION <FIRST.FUNCTION .S>>>

<SET I '1>

<SET PDL '(ROTATE.LF)>

<SET OUT.ILN.
     '((100 2)
       ((101 102 103 104 105 106 107 108 109 110 111 112)
        (1 2 3 4 5 6 7 8 9 10 11 12)))>

ENTERING AP.6.1. FROM LEVEL 1

<SET LEVEL '2>

<SET PDL '(ZZ IS.MOD.2)>

<SET J '1>

<SET PDL '(IS.MOD.2 IS.MOD.2)>

<SET IN.ILN
     '((100 2)
       ((101 102 103 104 105 106 107 108 109 110 111 112)
        (1 2 3 4 5 6 7 8 9 10 11 12)))>

<SET XF.ILN.
     '(((100 1) (100 2))
```

```
       ((1 2 3 4 5 6 7 8 9 10 11 12)
        (101 102 103 104 105 106 107 108 109 110 111 112)))>

<SET OUT.ILN. '#FALSE ()>

<SET OUT.ILN
     '((100 2)
       ((101 102 103 104 105 106 201 202 203 204 205 206)
        (2 4 6 8 10 12 3 5 7 9 11 1)))>
...

<SET J '7>

<SET PDL '(/S.MOD.2 ROTATE.LF)>

<SET IN.ILN
     '((100 1)
       ((1 2 3 4 5 6 7 8 9 10 11 12)
        (2 3 4 5 6 7 8 9 10 11 12 1)))>

<SET XF.ILN.
     '(((100 1) (100 2))
       ((1 2 3 4 5 6 7 8 9 10 11 12)
        (101 201 102 202 103 203 104 204 105 205 106 206)))>

<SET OUT.ILN.
     '((100 2)
       ((101 102 103 104 105 106 201 202 203 204 205 206)
        (2 4 6 8 10 12 3 5 7 9 11 1)))>

<SET OUT.ILN
     '((100 2)
       ((101 102 103 104 105 106 201 202 203 204 205 206)
        (2 4 6 8 10 12 3 5 7 9 11 1)))>

LEAVING AP.6. BECAUSE HAVE FOUND NEEDED FUNCTION


I think I have a function that will fill the bill.  It is:

<DEFINE AP.RESULT (S) </S.MOD.2 <ROTATE.LF .S>>>

To test the function, I shall apply it to the sample input and
see whether or not it yields the sample output that you gave
me.  If the function I came up with turns out not to be
applicable to the input, a MUDDLE error will result, and
MUDDLE will be in its listening loop.  If my function is
applicable, I shall display the sample input, the sample
output you gave me, and the output produced by (result
returned by) the function.  Yes, it checked.

<SET SMPL.IN '(A B C D E F G H I J K L)>

<SET SMPL.OUT '((B D F H J L) (C E G I K A))>

<SET TEST.OUT '((B D F H J L) (C E G I K A))>
END OF LAP.6 SCRIPT
```

An inherent and essential difficulty in automatic composition of functions from modules is delimitation of the search space. The procedure built into LAP is primitive: (a) make a few measurements of the sample input-output pair, (b) derive descriptors -- just five of them in the case of LAP.6 -- from the measures, (c) compare those descriptors with the descriptors associated with each function in the data base, (d) select, for the subset of the library to be searched, those functions whose descriptor patterns match the pattern of the sample-pair descriptors, and (e) search the subset exhaustively. As illustrated, the search is conducted in phases: first a search for a single function that will meet the requirement, then a search for a pair of functions, then a search for a trio, and so on. If there are M functions in the selected subset, and if the search is limited to singles, pairs, ..., and N-tuples, the worst case involves examining $M + M^2 + \ldots + M^N$ functions. How feasible that is depends, of course, upon the cost of examining a typical function, but more crucially upon the values of M and N.

To limit the cost of examining a typical function, LAP was designed not actually to apply a function to the sample input and test for a match to the sample output but, instead, to deal with surrogates of functions, as illustrated in the script. LAP deals with structures called "input outlines", "transfer outlines", and "output outlines". These structures abstract from inputs (arguments), functions, and outputs (results) the information that pertains to the subject of format with which LAP is concerned. In most instances it is far less time-consuming to apply a transfer outline to an input outline and test the resulting output outline than it is to apply a function to an argument and test the result.

In order to restrict M, the size of the searched subset of the library, it is desirable to have good procedures for selecting the subset. At present, LAP has only token procedures. Some of our current exploration of ideas about the matter are described in the next section.

2. How to Describe What Functions Do

It is essential both as a basis for retrieval of library functions, as discussed earlier, and as a basis for automatic programming, as discussed in the preceding subsection, to be able to describe what specific functions do when applied to arguments. The subject is deep, and the depths are full of difficulties, even impossibilities. It is closely related, of course, to the subject -- indeed, part and parcel of the subject -- of specifying what it is one wants a function to do when he requests a library or an automatic programmer to provide the functions. The technique of specifying by sample input-output pairs is seen to be severely limited when studied in any depth, but in fact it is nevertheless a very useful technique and a technique frequently employed in communication between human programmers and their clients. Our approach to

the behavioral description of functions is, in a similar spirit, to see how effective descriptions can be in practice and not to despair at the first appearance of an infinite set.

The descriptions with which we are currently working go only a little way beyond the descriptions of CALICO subroutines mentioned earlier. The current descriptions include:

1. Types and structures of arguments.
2. Other a priori objects (than arguments) upon which behavior of function depends.
3. Type and structure of result.
4. Other a posteriori objects (than result) that may be affected by application of function.
5. Relations between 3 and 1 and 2.
6. Relations between 4 and 1, 2, and 3.
7. Descriptors.
8. Category.
9. Functions that may be called by this function.
10. Library functions that call this function.
11. Atoms in this function that have global values that are not functions.
12. Atoms in this function that have local values.

In connection with items 1, 3, and 5, we are developing a language for description of structures and declaration of types. Let it suffice here to give the terms and three examples of expressions in a part (ALPHA) of the language. In each example, the description is given first in the description language and then in a more conventional notation. Brief explanations follow the trio of examples.

NOTATION OF DESCRIPTION LANGUAGE ALPHA

a,...,z, A, B, C, D	General names
E	ELEMENT
F	FORM
G, H	Variable integers
I	INTEGER
J, K	Variable (index) integers
L	LIST
M, N	Constant integers
O	OBJECT
P	ATOM
Q	QUANTITY (numeric)
R	REAL
S	STRUCTURE
T	STRING
U	UVECTOR
V	VECTOR
W	BOOLEAN
X, Y, Z	Variable real numbers
. (period)	Separator (same level)
/	Separator (go down one level)
_	Left arrow (result from argument)
• (center dot) or | (vertical bar)	Separator (toggle: enter/leave ALPHA)

1. •S/S $[[O_{11}\ O_{12} \ldots O_{1M_1}] \ldots \ldots [O_{N1}\ O_{N2} \ldots O_{NM_N}]]$

2. •VAB/AL3E.BU4I [(E$_1$ E$_2$ E$_3$) ![6 3 2 15!]]

3. LAb/bLN*GE_SAB/ASC.BSN/SGE

$$\underbrace{((C1\ C2 \ldots Cm)}_{A}\ \underbrace{(E_1\ E_2 \ldots E_N)}_{b}$$

$$\underbrace{((C1\ C2 \ldots Cm)}_{A}\ \underbrace{((E_{11}\ E_{12} \ldots E_{1G_1}) \ldots \ldots (E_{N1}\ E_{N2} \ldots E_{NG_N})))}_{B}$$

The first example describes a structure (i.e., any structured MUDDLE object) consisting of any number of structures consisting of any number and kind of objects.

The second example describes a vector consisting of two components, the first being a list of three (unstructured) elements and the second being a uniform vector of four integers.

The third example describes an operator that accepts the structure described by the right-hand side and returns the structure described by the left-hand side. The former is a structure consisting of two components, A and B, A being a structure of components C and B being a structure consisting of N structures, each consisting of some number G of elements -- with G having possibly N different values. The latter is a list of two components, A and b, A being the same as on the right-hand side and b being a list of $G_1 + G_2 + \ldots + G_N$ (signified by N*G) elements.

This is not the time to explain in detail the language, which is changing in the light of experience, but perhaps the illustrations will convey an idea of the objective: a compact notation for dealing with complex structured objects. A part (BETA) of the language not illustrated includes a much larger vocabulary of terms. Another part (GAMMA) is a set of default conventions dealing with relations between left-hand and right-hand sides of operator descriptions. We think that it will be possible to incorporate such relations into the descriptions in the most important and prevalent instances without significantly complicating the language. For example, 3 can be interpreted as implying that the elements E of the result are precisely the elements E of the argument, all arrayed in order in a single list instead of being subdivided into N substructures.

The most difficult components of the description of a function, it appears, are items 5 and 6:

5. Relations between result and arguments and other a priori objects.

6. Relations between a posteriori objects (other than result) and arguments, other a priori objects, and result.

We have explored several approaches and are developing a system that incorporates the following:

a. Use of a simple MUDDLE function as (part of) its own description. E.g., where "G" is "GREATER", "L" is "LESS", and "==" is "NUMERICALLY.EQUAL",

```
<DEFINE G==? (Q1 Q2) <NOT <L? .Q1 .Q2>>>
```

b. Use of a simpler but less efficient MUDDLE function as (part of) the description of a more complex but more efficient MUDDLE function that exhibits the same nontemporal behavior.

c. Use of one or more sample input-output pairs, as in LAP.

d. Use of the language ALPHA.BETA.GAMMA, described earlier.

e. Statement in a slightly modified MUDDLE of what is true of the world after the function has been executed.

f. Description in English, as terse as possible.

To expand slightly on e, consider a simplified version of the function INCR:, one that accepts as its argument an atom that has a local value, returns a number one greater than that value, and leaves the atom pointing to the resulting value. One can take care of the result by specifying that it is the same as would be returned by the (already described) function 1.GR. To describe the side effect, the fact that the atom (call it A) now points to a new (incremented) local value, however, it is necessary to write something like:

```
<TR <==? .A <1.GR <HOLD .A>>>>
```

That expression is interpreted by a metafunction that evaluates the HOLD, then executes INCR:, and then evalutes the remainder of the description. The "TR" refers to a function that can return "true" or "false", and it asserts that "TR" will return "true". Thus the description says that INCR: returns a value one greater than the local value originally pointed to by A and that, after execution, A points to that latter local value.

The approach just illustrated is derived, of course, from program verification. It appears that it will be useful, and perhaps less complicated, in program description. Even in the latter area, complexities arise, but they seem to be

tractable. One writes a set of MUDDLE functions, which need not be efficient, to test aspects of the state of the world. To describe a given function, then, he configures some of the test functions into a single complex test and asserts TR of its result. In describing dependencies of the a posteriori situation upon the a priori situation, of course, the HOLD function plays an essential role.

REFERENCES

1. Broos, Michael S., "Implementation of the VECTOR Data Type", SR.19.12, April 1973.

2. Burmaster, D.E., Karolyn Martin, and J.C.R. Licklider, "Convention II: Standards for Listings: Overview", GA.01.09.00.

3. Cutler, Scott E., "Computer Aided Evaluation and Design of Feedback Systems", S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1973.

4. Daniels, Bruce, and Ed Black, "MUDDLE Graphics User's Manual", SYS.11.04.

5. Daniels, Bruce, "MUDDLE Micro-Manual", SYS.11.03.

6. Data Types Special Interest Group, "Data Types for the Dynamic Modeling System", May 1972 (Memo).

7. Galley, Stuart W., "Debugging with ESP -- Execution Simulator and Presenter", SYS.09.01.

8. Haverty, Jack F., "Overview of Data Types", SR.19.10.

9. Haverty, Jack F., "Implementation of the String Data Type", SR.03.11.

10. Haverty, Jack F., Jeff Harris, and David Lebling, "Implementation of the Array Data Type", SR.19.07.

11. Hughett, Paul W., "Influence Nets: Mapping the Structure of a Process", S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1973.

12. Hughett, Paul W., and J.C.R. Licklider, "Convention II: Standards for Listings: Organization of the Standards", GA.01.09.01.

13. Knight, Frances, "Convention II: List of Convention II Documents", GA.01.00.

14. Lebling, P. David, "Printing Standard Data Types: DATCRP and DATPRT", SR.19.11.

15. Licklider, J.C.R., and Karolyn Martin, "Convention II: Overview of Glossaries", GA.01.02.00.

16. Licklider, J.C.R. and Karolyn Martin, "Convention II: Glossary of Standard Notation", GA.01.02.01.

17. Licklider, J.C.R., and Karolyn Martin, "Convention II: Glossary of Standard Terms", GA.01.02.02.

18. Licklider, J.C.R., and Karolyn Martin, "Glossary of Abbreviations and Expansions", GA.01.02.03.

19. Licklider, J.C.R., "Convention II: Standard MSR Headers, Data-Set Headers, and SDAT/USDAT Triads", GA.01.03.

20. Licklider, J.C.R., "Convention II: Data Types", GA.01.04.

21. Licklider, J.C.R., "Convention II: Standards for Listings: Remainders", GA.01.09.03.

22. Licklider, J.C.R., "Convention II: Subdivision of Packages", GA.01.11.

23. Licklider, J.C.R., "Convention II: DSRs, QSRs and RSRs", GA.01.12.

24. Licklider, J.C.R., "Convention II: Miscellaneous Update Information", GA.01.14.

25. Licklider, J.C.R., "Convention II: Design of Data Sets", GA.01.15.

26. Licklider, J.C.R., "Convention II: The Data System", GA.01.16.

27. Liu, Mark H., "DETAIL: A Graphic Debugging Tool", B.S. Thesis, Department of Electrical Engineering, M.I.T., February 1972.

28. Martin, Karolyn, "Convention II: Standard Format for DG System Documents", GA.01.01.

29. Martin, Karolyn, "Convention II: Standards for Naming Files", GA.01.05.

30. Michener, James, et al, RFC #493, "Graphics Protocol", NIC #15358.

31. Pfister, Greg, "A MEDDLE Manual", SYS.11.02.

32. Pfister, Greg, "A MUDDLE Primer?", SYS.11.01.

33. Reeve, Chris, "ISR, MSR, QSR, RSR and DS Macros", MCR.01.04 (Draft).

34. Reeve, Chris, "Convention II: How to Use MACRO TS to Conform to Convention II", GA.01.06.

35. Reeve, Chris, Marty Draper, D.E. Burmaster, and J.C.R. Licklider, "Convention II: Standards for Listings: Listing Abstracts", GA.01.09.02.

36. Reeve, Chris, and J.C.R. Licklider, "Convention II: How to Take the First Step into the New World of CAREDL", GA.01.13.

37. Reeve, Chris, "Implementaion of Location-Insenstive SRs Using the OFFSET Pseudoinstruction", GA.01.17.

38. Reeve, Chris, "Taking the Second Step into the World of Pure DYNAL", GA.01.18 (Draft).

39. Seriff, Marc S., "Virtual File Management Service for the ARPA Network", S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1973.

PROGRAMMING TECHNOLOGY

PUBLICATIONS

1. Black, Edward H., "Computer Graphics", Cross Talk, Vol.2, No. 3, Department of Electrical Engineering, M.I.T., December 1972, p. 3.

2. Galley, S. W., "PDP-10 Virtual Machines", Proceedings of Workshop on Virtual Computer Systems, ACM SIGARCH-SIGOPS, Harvard University, March 1973, pp. 30-34.

3. Licklider, J. C. R., "Consumer(Communication) Networks for Computers", RCA/MIT Research Conference, RCA Engineer, Vol. 18, No. 5, February/March 1973, pp. 69-70.

AUTOMATIC PROGRAMMING

Academic Staff

Prof. M. L. Dertouzos
Prof. G. A. Gorry
Prof. C. Hewitt
Prof. B. Lisbov
Prof. S. E. Madnick
Prof. W. A. Martin
Prof. J. Moses
Prof. J. Weizenbaum

R. J. Fateman
V. S. Pless
P. S-H. Wang

DSR Staff

E. R. Banks
R. A. Bogen
J. P. Golden
J. P. Jarvis
R. Schroeppel

J. M. Shah
A. Sunguroff
D. C. Watson
J. L. White

Graduate Students

S. L. Alter
R. V. Baron
V. A. Berzins
P. B. Bishop
G. Brown
J. S. D'Aversa
A. C. England
S. P. Geiger
M. J. Ginzberg
D. L. Isaman
P. Jessel
R. B. Krumland
J. Kulp
T. Landau
M. Laventhal
C. Lynn
W. S. Mark
M. L. Morgenstern
G. Pfister
M. Rashwan
G. Ruth
R. J. Steiger
L. Tsien
S. R. Umarji
T. Victor
S. A. Ward
J. Wish

Graduate Students (cont.)

R. T. Wong

D. Yun

Undergraduate Students

J. S. Adamczk

H. I. Badlan

G. G. Benedict

T. L. Davenport

R. Law

D. J. Littleboy

S. M. Macrakis

W. E. Matson

B. Niamir

G. L. Peskin

R. T. Petraitis

J. P. Reese

E. C. Rosen

S. E. Saunders

R. M. Siegel

G. L. Steele

B. M. Trager

R. E. Zippel

Support Staff

J. S. Lague

N. J. Robinson

Guests

A. Miola

J-t. Wang

H. Wantanabe

AUTOMATIC PROGRAMMING DIVISION

INTRODUCTION

The objective of the Automatic Programming Division is to develop fundamentally new software technology for the programming and use of computers in practical applications such as business data processing, medical diagnosis, symbolic applied mathematics, automatic control, and management decision systems. We are attempting to do this not through the construction of a single software system, but through the development of several prototype systems, each designed to explore solutions to one or more problems faced in current programming practice. A number of these systems and their results are described below.

Since this is the first year the division has existed, our goal has been to bring as many programs as possible to the point of a simple demonstration. In the coming year some of these will be revised, others extended, and some abandoned. A number of new faculty and senior research people will be joining us next year; they will certainly have ideas of their own. As these are also tested we expect to gain the confidence to build larger systems as we have done in algebraic manipulation. In that area our MACSYMA system now has become one of the largest and most sophisticated applications systems available and has been used very successfully in the past year.

The division is currently broken down into four groups: Automatic Programming, Mathlab, Medical Decision Making, and Engineering Robotics. We have included separate reports for each group. It may also be helpful to summarize the state of the division as a whole.

The division has enhanced its computational resources. The MATHLAB PDP-10 has been expanded and the software upgraded to support multiple users of large programs. A good LISP has become operational on MULTICS, and the Engineering Robotics Group has developed software for the PDP-11/45.

A number of applications suitable for research have been identified in mathematics, medicine, control, and management.

Research is being done on the automatic scheduling and allocation of computational resources, improving the ability of computer programs to explain what they are doing, acquiring problem descriptions from users, and the development of formalisms for describing the knowledge possessed by experts to machines so that the machines can use it effectively in solving problems.

AUTOMATIC PROGRAMMING GROUP

A. INTRODUCTION

The focus of the Automatic Programming Group is the application of computers to domains where much is already known about how to solve a given problem, yet the great variety of specific problem contexts which arise has so far made it impossible to write a single computer program which would apply in every situation. The basic idea underlying our attack on this issue is to represent the knowledge about how to solve problems in the given domain at a higher level of abstraction than is currently done. This abstract knowledge is then used to generate a program for any specific user context.

Management data processing, and information and decision systems, provide a good example of the type of problem domain we have in mind. Figure 1 shows a classification of some software packages currently offered for sale by IBM. These fall roughly into two categories:

a) Support of higher level decision making through the selective retrieval of data and the application of extremely simple models (mathematical in character).

b) Automation of daily operational procedures through the incorporation of knowledge about how to perform these procedures into computer programs.

Although such systems may perform well in the environment for which they were designed, it is difficult to adapt them to a new environment if changes other than changes in parameter values are required. This is particularly true of the operational level programs which contain knowledge of particular business procedures.

A step toward improving this situation has been taken by the IBM System 3 Application Customizer and similar programs offered by other firms. The customer is given a long multiple choice questionnaire. A typical question might be "When a customer transaction is processed, the computer can compare the amount he owes to whatever credit limit you assign him and print a note if the amount due is over the limit. Should this be done?" The answers to these questions are used to select precoded program segments and assemble them into the user's program. This approach provides much greater flexibility than the precoded program products, and it also provides the user with a structure for the decisions he must make, suggesting the standard choices. However, the user does not have any way to specify a procedure which is not incorporated into the questionnaire and there is no way for the system to automatically alter the data structure used in the solution to be more efficient or more compatible with other uses of the same data.

We are constructing a system, Protosystem I, which will go

Structured Problems	Structured Support of Unstructured Problems	Unstructured
Planning	1) Planning systems generator 2) Public utility financial planning system	
Management Control	1) Project management system IV 2) Aerospace info. and control system, project scheduling, budgeting, evaluation, and control	
Operational Control 1) Agribusiness Mgmt. info. sys. 2) System/3 bill of material processor 3) OS/360 inventory control 4) System/360 order allocation system 5) OS/360 requirements planning 6) Advanced life info. system 7) Shared hospital accounting system 8) Shared laboratory info. system 9) IBM basic courts system 10) Consumer goods system forecasting 11) PALIS automobile-homeowners	1) Dynamic shop floor control 2) Capacity planning-finite and infinite loading 3) Consumer goods system-allocation	

Figure 1.

Classification of IBM "Program Products"

considerably further in providing user flexibility. This system is shown schematically in Figure 2. As with the Customizer, the user's interaction will begin with a questionnaire, but in Protosystem I, it will be interactive. The nature of the questions, however, will be altered from asking for choices of procedure to asking for information describing the customer's environment. The questionnaire will not require that the user give multiple choice answers: instead, constructive responses will be allowed. How, then, can we be sure that the system "understands" the user's problem? We have constructed a relational modeling language, MAPL, in which we can construct a general model of the environment of a business procedure such as billing or order allocation. We require that the user's problem be an instantiation of this general model. (MAPL is described in section I.) This area of problem acquisition is one we will be exploring further in the future.

Once a description of the user's problem has been acquired, the system guides him in the construction of a solution in the form of a block diagram. We have not yet implemented this key part of the system except in very elementary form. However, the Ph.D. thesis of G. Sussman, who will be joining Automatic Programming next year, contains many of the techniques which we will need for a full implementation.

Once a solution has been found, there is a possibility that it is not what the user wants. The user may have mis-described his problem or he may have made a bad decision on some aspect of the solution where he did not follow the system's advice. One way the user can gain confidence in the solution is to explore its behavior through simulation. In section II we describe a program which not only simulates the user's solution but then attempts to explain the difference between the simulation and what the user expected by making deductions about the model.

It will also be useful if the user can ask questions about the system's knowledge and its solution using a subset of English. A program for translating from English to MAPL is described in section III. Having the ability to query MAPL models with English also allows us to experiment with systems which give more support to management decision makers.

Finally, Protosystem I will translate the user's solution from block diagram form into PL/I. Section IV traces a sample problem through this translation.

B. MAPL

MAPL is a language for building relational models of the world. It is not yet complete; in particular the facilities for quantification and for describing how to make deductions are still evolving. The use of relational models has become popular in file design, artificial intelligence languages, and psychology. We hope MAPL will evolve into a useful implementation of the best ideas and will allow subsequent researchers to build on the work of others.

In MAPL, the world is considered to be made up of a collection

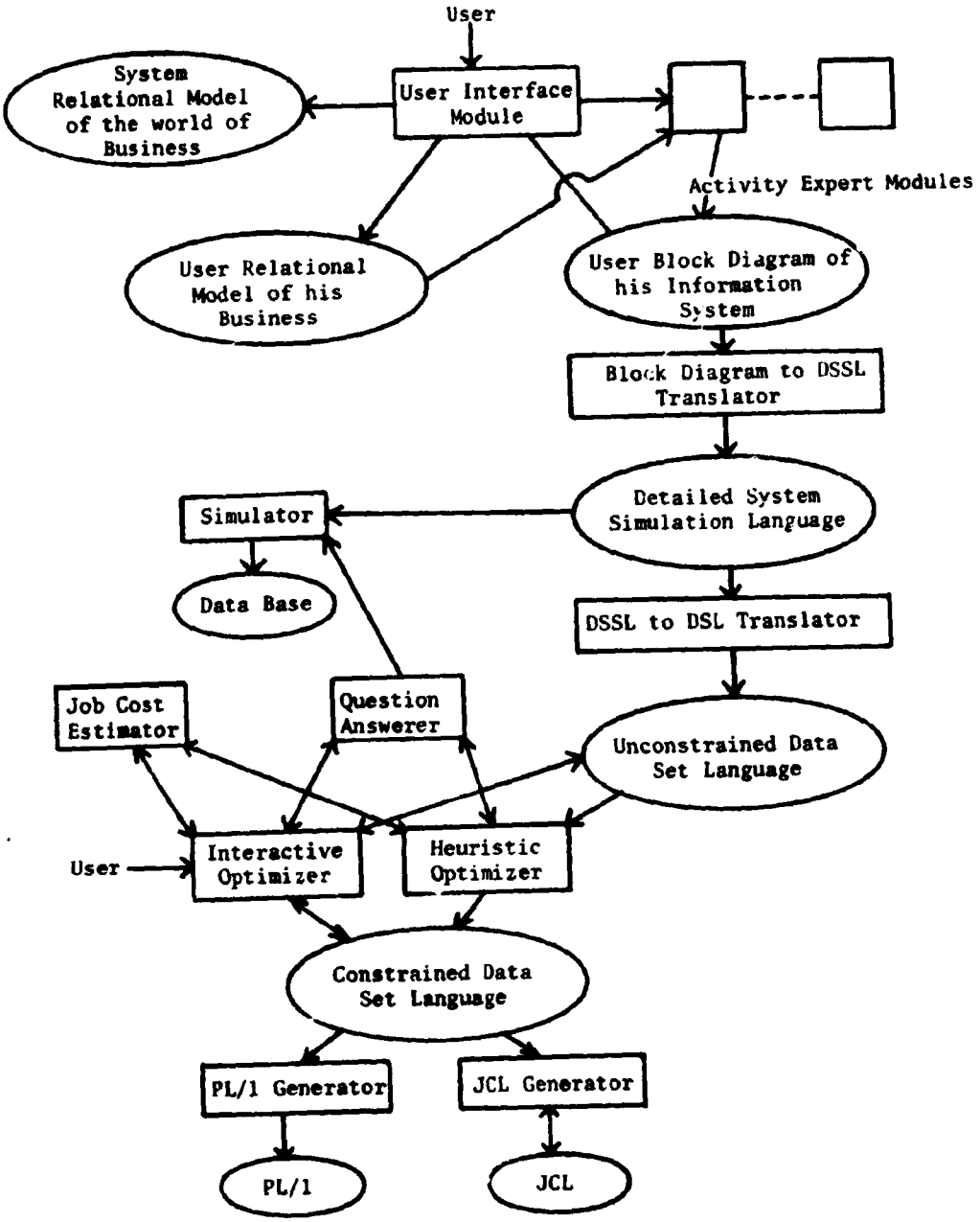


Figure 2.
Protosystem I

of objects. These objects are divided into subsets, such as the subset of all objects which are fruit. The subset of all objects which are fruit is represented by %FRUIT, a predicate which is true only for objects in this subset by #FRUIT, and a typical object of this subset by $FRUIT. A subset of %FRUIT might be %APPLE. We state this in MAPL as A-K-O (A-K-O APPLE FRUIT). A-K-O is read "a kind of." FRUIT and APPLE are referred to as concepts. Since one concept can be a kind of several other concepts, the concepts form a lattice under set inclusion.

It is interesting to ask how many concepts a world model might contain. This can be approached by counting the number of distinct words in technical questionnaires, books, and case studies, and by building world models. We guess that interesting models can be built with less than 10,000 concepts, although the models we have actually built have all contained only a few hundred.

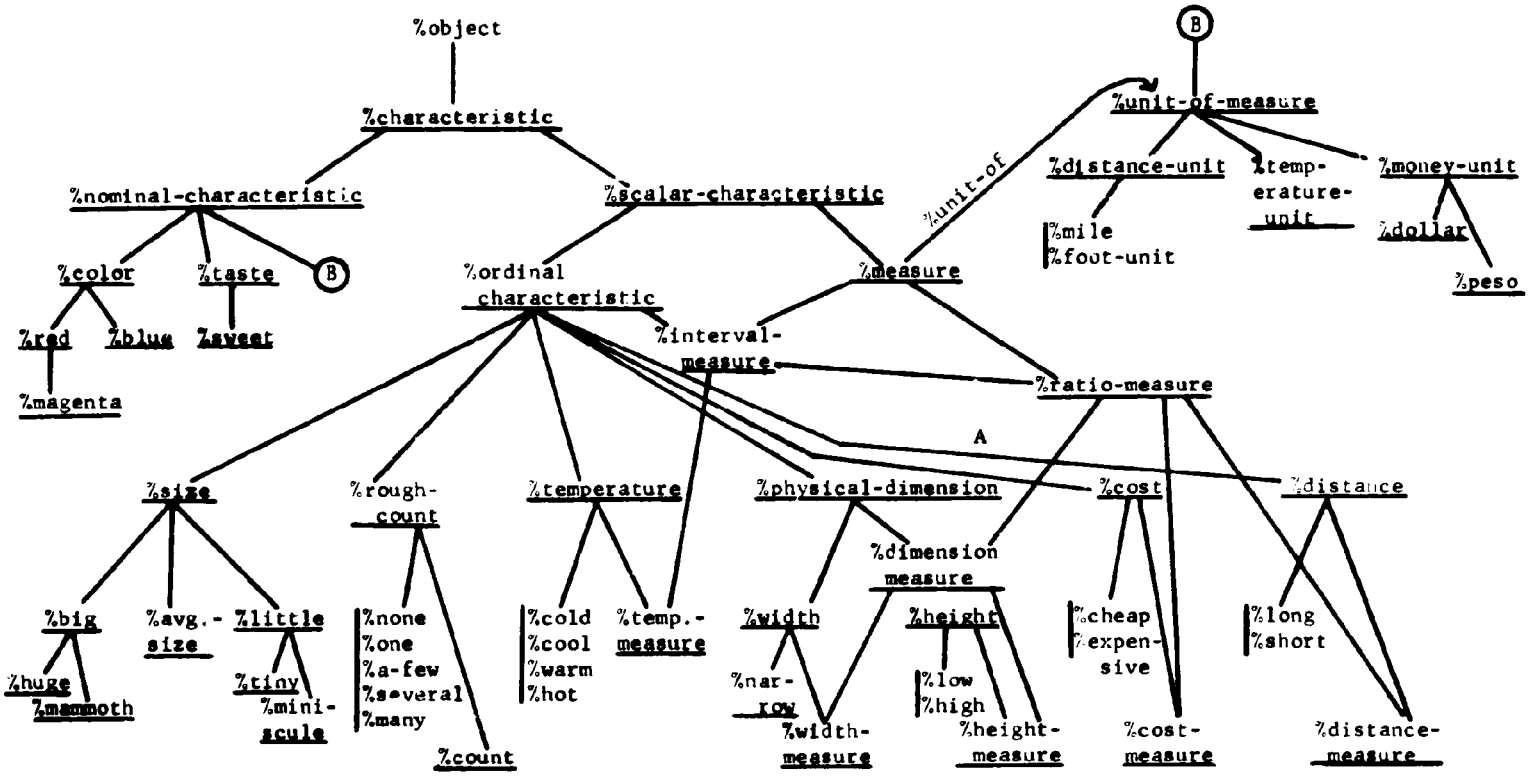
A MAPL world modeler can assign properties to concepts (actually to the #, %, $ manifestations of a concept). For example, to state that any fruit can have color he would write (A-R-O COLOR-OF COLOR FRUIT). This declares that the relation COLOR-OF takes tuples, the first member of which is a COLOR and the second member of which is a FRUIT. Suppose that during problem acquisition the user attempts to state that his apples are red. Given the above declaration of COLOR-OF, the system will accept this if RED is a kind of a color and the user's APPLE is a kind of a fruit.

It is our goal to find standard methods of handling time, location, characteristics, and other relationships in MAPL, so that the individual world modeler will not have to work this out. Figure 3 shows a classification of characteristics as nominal, ordinal, interval measure, or ratio measure. Values of nominal characteristics cannot be ordered. One does not say that red is greater than blue. Values of ordinal characteristics can be compared, but no unit of measure exists. When the world modeler defines a new characteristic he should say what class of characteristics it is a kind of. The system will then automatically know certain things about it.

MAPL has a number of other features not described here. For example, any relation or tuple can participate in another relation or tuple. An example of its use in making deductions is reported by the Medical Decision Making Group.

C. DEBUGGING MODELS

A key part of programming is the creation of the model of the problem to be solved and the model of the proposed solution. Any model will be an approximation of the real situation, and even if it is consistent within itself we must rely on the user to evaluate its suitability. The problem of internal consistency is also, in general, unsolvable within the system. We can, however, gain confidence that the model is all right by exercising it and checking to see if it meets certain expectations set by the user.



All links are %A-K-O unless indicated.

Figure 3.

Partial Concept Net for Characteristics.

Suppose a user creates a model and asks the machine to do a simulation with it. The user also describes what he expects the results of the simulation to be. If the actual results differ from these, they may represent the manifestation of some bug, or incorrect description or decision, in the model. We have a program which attempts to locate possible bugs. At present it knows about competition for resources, and time sequencing problems of the type which occur in business games.

Suppose the user presents the program with the following tiny model:

> "Consider the following model of sales. A sale is a probabilistic occurence which depends only on the amount of advertising done. Advertising costs $3,000 per page and is good for one quarter. I buy three pages of advertising per quarter if the money is available. Sales take place during sales calls on customers. There is one call per salesman per quarter: a customer never buys more than one unit. If a unit is sold, the company records $5,000 in accounts receivable which is not collected for another two quarters. At any time, any salesman has a 5% chance of quitting. If a salesman quits, a new man is hired. After three months of training, this man becomes a salesman and may start making calls. Both salesmen and trainees are paid $1,000 per quarter. Trainees also have a 5% chance of quitting at any time."

The user would input this model into the program with the program's model specification language (MSL). In these terms, the model looks like:

```
(*ACTIVITY HIRING
     (*PREREQUISITES (*PRESENT (1000 CASH)))
     (*SCHEDULE (ON QUIT))
     (*PRIORITY 2)
     (*OUTPUT '(A TRAINEE))
)
(*ACTIVITY ADVERTISING
     (*PREREQUISITES (*PRESENT (3000 CASH)))
     (*SCHEDULE 3)
     (*TAKES 1)
     (*PRIORITY 3)
     (*OUTPUT '(1 PAGE-OF-ADVERTISING))
)
(*ACTIVITY TRAINING
     (*PREREQUISITES
          (AND
               (*PRESENT (1000 CASH))
               (*PRESENT (SOME TRAINEE))
          )
     )
     (*TAKES 3
     (*OUTPUT '(A SALESMAN))
)
(*ACTIVITY SALES-CALL
     (*PREREQUISITES
          (AND
```

```
                    (*PRESENT (1000 CASH))
                    (*PRESENT (1 UNIT))
                    (*PRESENT (SOME SALESMAN))
               )
          )
          (*TAKES 1)
     )
     (*ACTIVITY A-R-MATURATION
          (*PREREQUISITES (*PRESENT (5000 A-R)))
          (*TAKES 2)
          (*OUTPUT '(5000 CASH))
     )
     (*EVENT SALE
          (*CONDITIONS SALES-PROBABILITY)
          (*ACTIVITIES (SALES-CALL)
               (*OUTPUT '(5000 A-R))
          )
     )
     (*EVENT QUITTING
          (*CONDITIONS (UNIFORM .05))
          (*ACTIVITIES (SALES-CALL)
               (*CANCEL)
               (*REMOVE '(THAT SALESMAN))
          )
          (*ACTIVITIES (TRAINING)
               (*CANCEL)
               (*REMOVE '(THAT TRAINEE))
          )
     )
     (*FUNCTION SALES-PROBABILITY
          (*ARGUMENTS (ALL PAGE-OF-ADVERTISING))
          (*RETURN (AD-FUNCTION))
     )
```

(We will not show the exact nature of AD-FUNCTION, as it is of no importance to the example. Note that A-R denotes "accounts receivable" throughout the model.)

In MSL a model is described as a collection of ACTIVITIES. Each ACTIVITY has certain properties. For example, the first activity above, HIRING, requires $1,000 cash, is done when a salesman quits and produces a trainee. In competition for resources it has a priority of 2.

Now suppose the user gives the program the following:

```
(*SIMULATE 4 1
     (30000 CASH)
      (50 UNITS)
      (DON SALESMAN)
      (MARK SALESMAN)
      (STEVE SALESMAN)
      (BILL SALESMAN)
(*WANT 6 SALE)))
```

which states that a simulation of 4 time periods with the initial conditions of $30,000, 50 units, and four salesmen should result in six items sold. The results of the simulation are shown in

Figure 4. Only 5 units rather than 6 were sold. The program now attempts to determine why sales were low by setting a goal of increasing sales one unit in time period 4. This goal will lead to other subgoals. For each goal and subgoal the program uses the model and the simulation history to ask two questions.

(1) Why didn't you meet this goal before?
If there is no good reason,
(2) How could we do this?

A line of reasoning which might be followed by the program is indicated by Figure 5. From the model it sees that one way to increase sales in period 4 is to increase the probability of a sale on each sales call. This can only be done by increasing advertising. The normal three pages of advertising wasn't done, however, because we were short of cash in period 4. We could have more cash in period 4 if we could generate more accounts receivable in period 2. To do this, we need more sales in period 2. We can get more sales either with more salesmen or more advertising. However, the training period precludes getting more salesmen by period 2. This leaves us with the possibility of buying more than 3 pages of advertising in period 2.

We do not claim that this solution should be adopted, but we feel that it will be useful to present the user with this line of reasoning. Because it doesn't have all the facts, the program's conclusions may be entirely inappropriate to the situation, but the line of reasoning may show the user that he has given the program an inadequate model or it may remind him of a facet of the problem which he ignored. While it may be very difficult to make a program which is an authority on models, it may be possible to make one which has interesting comments to make. There are many important general concepts, such as feedback, which the program does not yet understand. There are many models here at M.I.T. which have been found useful in business situations, and which can be used as examples in expanding the program.

D. ENGLISH LANGUAGE INPUT

We feel that English language input will be important both in allowing us to obtain the knowledge of experts and in supplying expertise to the general public. We examined a number of the existing English input routines and found that while many good ideas had been discovered, no existent program lent itself to extension as a general purpose routine. We have designed a new routine and implemented it in full generality for one test sentence:

"How much did we sell to Sears in '72?"

A description of the program's behavior on this sentence is given below.

One of the new features of the parser is the use of a case grammar. The basic tenet of case grammar is that the sentence consists of a verb and one or more noun phrases (or other sentence elements) each associated with the verb in a particular relationship. This view is useful in analyzing the sentences:

CASH	A-R	SALESMEN	TRAINEES	UNITS ON HAND	SALES OF UNITS	SALES CALLS	PAGES OF ADV
$30,000	$0	4	0	50			
$17,000	$10,000	4	0	48	2	4	3
$5,000	$15,000	3	1	47	1	3	3
$2,000	$10,000	3	1	46	1	3	3
$0	$10,000	3	1	45	1	3	1

Figure 4.

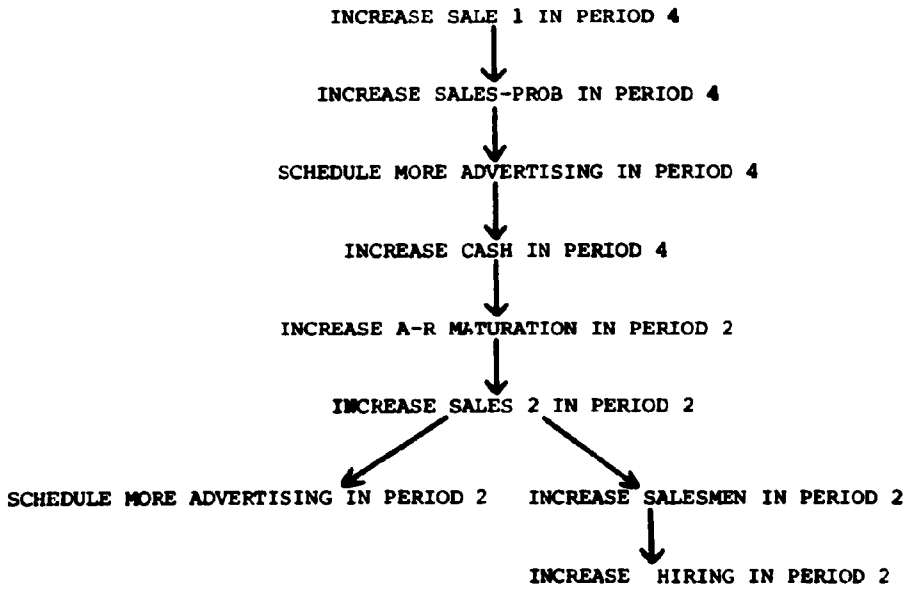


Figure 5.

1) John opened the door with a stick.

2) A stick opened the door.

3) The door opened.

In 1) we take John as the agent, a stick as the instrument, and the door as the object. Sentences 1), 2) and 3) show how a verb like open takes the agent, object, or instrument as the surface subject. Our scheme involves listing, for each verb meaning, what cases it takes and what predicate a noun group or other construction has to pass in order to be acceptable for that case. We must also list for each verb meaning what cases a noun group could be depending on its position in the sentence or the preposition which precedes it. The verb meanings are arranged in a MAPL lattice which reduces the redundancy in the specification.

The parser recognizes the following aggregates of words and phrases:

MAJOR-CLAUSE

SECONDARY-CLAUSE

NOUN-GROUP

ADJECTIVE-GROUP

ADVERB-GROUP

QUESTION-GROUP

PREPOSITION-GROUP

VERB-GROUP

It builds up a MAPL expression for each phrase. The MAPL expression corresponding to "How much did we sell to Sears in '72?" is shown in Figure 6. A finite state transition network has been written for each phrase. Each state can have three kinds of arcs leading out of it: next-unit, try-branches-of (indicated by -----→ in the word order charts below), and no-success. When building the MAPL expression corresponding to a phrase, the parser tries each of the next-unit arcs out of the current state of that phrase; if none of these applies it looks for the try-branches-of arc (of which there is at most one) and tries the arcs of the state indicated by it. If none of these leads to success it looks for a no-success arc, which indicates under what conditions the phrase can be complete without further constituents added. Each arc gives the syntactic type of the word or phrase which must be found next and a function which must be successfully applied to the MAPL expression built up so far, and the MAPL expression for the phrase just found. If the function is successful, it returns the new partial MAPL expression for the part of the phrase found so far.

For example, a fragment of the noun group network currently implemented looks like:

```
(NOUN-GROUP   DET=NUM=ADJ=NOUN=PRONOUN
 (WE=YOU    ADD-PRONOUN-TO-NG
    (NOUN-GROUP    NUM=ADJ=NOUN=ALL
         (NO-SUCCESS    DO-NOTHING1   SUCCEED)))
(TRY-BRANCHES-OF   DO-NOTHING1   ORDINAL-SUBTREE))

     GO166
       AGENT-OF GO166
          TYPE-OF PRONOUN-NG
          SYNTHETIC-CASE-OF SUBJECTIVE
          PERSON-OF FIRST
          NUMBER-OF PLURAL
     OBJECT-OF
          TYPE-OF QUESTION-NG
          COUNTABILITY-OF MASS
          RELATION-QUESTIONED-OF COUNT-OF
          NUMBER-OF SINGULAR
     TIME-OF GO175
          TIME-REFERENCE-OF IN GO166 GO174
             A-K-O YEAR-1972
             T    -OF OBJECT-NG
             NUMBER-OF SINGULAR
          RECIPIENT-OF GO172
              A-K-O SEARS
              NUMBER OF SINGULAR
          RELATION-QUESTIONED-OF OBJECT-OF
          TENSE-OF (PAST)
          A-K-O SELL-GOODS
          PERSON-NUMBER-OF PLURAL
          TYPE-OF WH-QUESTION-CLAUSE
```

Figure 6.

Output of the parser for the sentence
"How much did we ell to Sears in '72?"

The first line says that we have a NOUN-GROUP going and we are currently looking for something which is a kind of DET=NUM=ADJ=NOUN=PRONOUN. The second line says that if we in fact find something which is a kind of WE=YOU then we attempt to apply the function ADD-PRONOUN-TO-NG to the MAPL form of the noun group, and the MAPL form of WE=YOU. If this function returns NIL the parse can't proceed. The only alternative is then given by the fifth line, which says that if the function DO-NOTHING1 can be applied to the NOUN-GROUP MAPL form with a non-NIL result we can try the branches of the node named ORDINAL-SUBTREE. If ADD-PRONOUN-TO-NG is successful, the third line tells us that we then have a noun-group going and are looking for a NUM=ADJ=NOUN=ALL. If we don't find one, the fourth line says that if DO-NOTHING1 applied to the NOUN-GROUP MAL form is non-NIL, then that result is the completed noun group, which can then be added to a superior group or clause.

During the parse, the parser maintains a stack of pairs: a current state in a phrase and a partial MAPL expression. The stack is started off with one pair; the first state of MAJOR-CLAUSE and a null MAPL expression. The parser then looks at the next word of the input string and takes a number of actions which are dependent on our view of the structure of English. First, it checks to see if the word starts a noun idiom or proper noun expression and builds it if it does. Failing this, it tries to add the word to the current phrase. Failing this, it checks to see if the word would begin one of the other phrases. If it will, it starts that phrase. It then checks to see if this new phrase could possibly be fitted onto the current one when the new one is finished. It does this by comparing what we have going in the new phrase with what we are looking for in the current one. If the new phrase can yield a constituent we are looking for, it adds the new phrase to the stack. When a phrase is finished the parser removes it from the stack and tries to add it to the one immediately above. If this fails, it checks to see if the one above can be considered complete without additional constituents being added. For example, consider:

I rode down the street in the car.

At some point we will in effect have

I rode →

down →

the street →

in the car.

The parser will try to form

I rode →

down →

the street in the car

but the MAPL world will block this. The parser will then form

I rode →

down the street

in the car.

and then it will form

I rode down the street →

in the car.

and it will then be successful in attaching in the car to the main clause. In starting a new group the parser must also consider the possibility that it begins a secondary clause. For example,

We celebrated the day the rain came.

The parser will get

We celebrated →

the day →

and it will then see that the next word starts another noun group. A noun group cannot post-modify a noun group, but it can start a secondary clause. The parser forms

We celebrated →

the day →

→

the →

and continues as normal. All parsings are found by taking all branches at early decision points and the corresponding MAP expressions generated for each. Complete constituents are saved so that they are not generated twice by different parses. Negation, surface-objective-case, and person-number are not used to stop a phrase until it is time to add it to the one above. Such features don't seem to block many false parses. That is, these features are checked by the functions which combine MAPL forms rather than being used to describe what we are looking for and what we have going. As the parser finds the noun groups of the clause from left to right it is not always able to assign them to the proper case immediately, therefore it holds them until enough is known. For example, consider the sentence,

How much did we sell to Sears in '72?

The parser attempts to move through the sentence putting constituents aside (but remembering their position) until it finds the surface object. First it finds "how much" and remembers this as the first noun group. Then it finds "did" and remembers this as a possible auxilliary. Next it finds "we" and remembers this as the second noun group. Then it finds "sell". It now knows that "we" is the surface subject and "did sell" is the verb. Next it finds "to Sears". Since this is a preposition group it knows there are no surface objects. It now considers each meaning of "sell". For each meaning it looks up the possible cases for the surface subject and discovers that "we" could be either the agent or the object. Currently it does not attempt to discover that "we" is Globe Union Battery Company; but that would not change what follows. It discovers that "we" passes both the object and agent predicates for sell, so it remembers that these two possibilities remain, and proceeds with the parse. It finds "to Sears". It finds that "to" flags the recipient for "sell" and that the recipient does not take a prepositional phrase in fact, but only the object. "Sears" passes the predicate for recipient and is assigned. The parser finds "in '72". "In" flags time, which does take the whole prepositional phrase. "In '72" passes the predicate for time and is stored as

(TIME-REFERENT-OF IN Major clause YEAR-1972).

Now the sentence is finished and "How much" has not been needed by a dangling preposition. Since it occurs in first position, "How much" must thus be the object; this makes "we" the agent.

E. TRANSLATION INTO PL/I

As shown in Figure 2, in Protosystem I, the model of the solution to a user's problem is expressed in Detailed System Simulation Language (DSSL). To get an idea of the nature of this language, consider the A&T Supermarket Micro case shown schematically in Figure 7. Each day stores order from a central warehouse. The warehouse fills the items from inventory and then orders items which are in short supply from a supplier. The supplier fills the orders the next day and the warehouse updates its inventory. Protosystem has generated PL/I for this example. An inventory file of 4,000 items is kept at the warehouse. The quantity of each item on hand at the warehouse, taking into account receipts from suppliers, is given by

```
BEGINNING-INVENTORY (DAY, ITEM) =

  IF  DEFINED  (FINAL-INVENTORY  (DAY -- 1, ITEM))

    AND  DEFINED  (QUANTITY-RECEIVED  (DAY, ITEM))

   THEN   FINAL-INVENTORY  (DAY  -  1, ITEM) +

              QUANTITY-RECEIVED  (DAY,   ITEM)

  OR  IF  DEFINED (FINAL-INVENTORY  (DAY  -  1, ITEM))
```

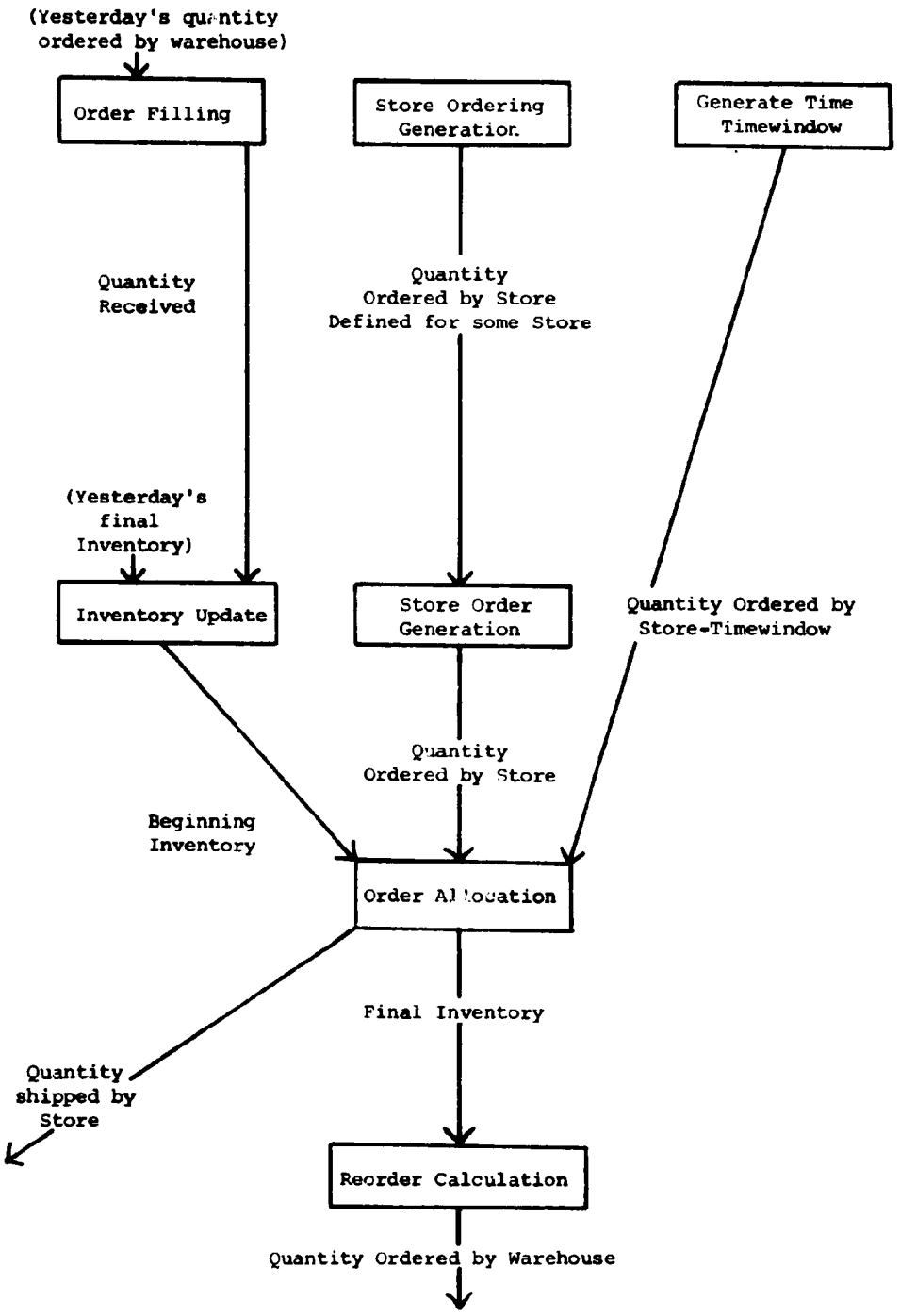


Figure 7.
A&T Micro Case

THEN FINAL-INVENTORY (DAY - 1, ITEM)

ELSE UNDEFINED

BEGINNING-INVENTORY is to be calculated for each item each day. As shown in Figure 6, the name of this calculation is Inventory Update.

BEGINNING-INVENTORY (DAY, ITEM) and QUANTITY-ORDERED-BY-STORE (DAY, STORE, ITEM) are inputs to the Order Allocation Calculation. This calculation is an aggregate operation on QUANTITY-ORDERED-BY-STORE because on a given day it must allocate a given item across all stores ordering it. In DSSL all such aggregate operations are represented by special functions built into the language. We have currently defined SUBSET-ALLOCATE, SUBSET-COUNT, SUBSET-MAX, SUBSET-MIN, SUBSET-NUMBER, and SUBSET-PLUS. It is our assumption that most business data calculations which involve aggregation can be expressed in terms of a rather small set of such functions.

By way of illustration we will define SUBSET-PLUS before returning to the allocation of warehouse items. SUBSET-PLUS is defined by the expression

$$\text{Output (period, } k_1\ k_2 \cdots k_i\ k_{i+1} \cdots k_n) = \sum_{k_i} \text{Input (period, } k_1 \ldots k_n)$$

where Input, Output, and k_i are given to SUBSET-PLUS as parameters.

Our current version of SUBSET-ALLOCATE says that store orders are filled in any sequence. Each store receives the amount it ordered unless not enough of the item is remaining, then it receives none. SUBSET-ALLOCATE takes as inputs the orders and the beginning inventory and produces as outputs the quantity shipped and the final inventory. For A&T Micro we have

Order Allocation

SUBSET-ALLOCATE(

QUANTITY-SHIPPED-TO-STORE (DAY, STORE, ITEM)

FINAL-INVENTORY (DAY, ITEM)

QUANTITY-ORDERED-BY-STORE (DAY, STORE, ITEM)

BEGINNING-INVENTORY (DAY, ITEM)

ITEM)

After the value of FINAL-INVENTORY has been computed for each item, Reorder Calculation determines if the warehouse should order more of the item from the supplier. The amount to be ordered is defined by

```
QUANTITY-ORDERED-BY-WAREHOUSE (DAY, ITEM) =

IF      DEFINED (FINAL-INVENTORY (DAY, ITEM))

     AND    FINAL-INVENTORY (DAY,ITEM) < 100

        THEN 1000

ELSE UNDEFINED
```

Inventory Update, Order Allocation, and Reorder Calculation are the three calculations which are to be implemented in PL/I on the warehouse's computer. The other calculations in Figure 6 are needed only to define and describe the input variables QUANTITY-RECEIVED and QUANTITY-ORDERED-BY-STORE.

As shown in Figure 2, the DSSL is translated into DSL. Each of the variables is converted into a DSL data set. In this representation, each of the other parameters of the variable except the period becomes a key and the variable becomes a data value. For example,

QUANTITY-ORDERED-BY-STORE (DAY, STORE, ITEM)

becomes a data set with fixed length records of the form

data value	QUANTITY-ORDERED-BY-STORE
key	STORE
key	ITEM

In this example, it will not be necessary to consider the possibility that the values of keys are the output of a computation. We will define ITEM to take on all values in the set SET-OF-ITEMS and STORE to take on all values in the set SET-OF-STORES. The DSSL description will tell us how many elements each of these sets contains and give a predicate which is true only for members of the set.

Suppose there are s elements in the set SET-OF-STORES and i elements in the set SET-OF-ITEMS, then there are s times i possible records in the data set QUANTITY-ORDERED-BY-STORE. We will make the convention that a record is physically present in a data set only if its data value is defined. It is important to know the number of records to be expected in a data set on a given day in order to optimize the PL/I programs for the high volume operations. To this end, we will define the key predicate of a data set to be a predicate on the time and the keys of a data set, which is true if the data set physically contains a record with those keys at that time. Clearly, the number of records in a data set can vary with time. (Stores don't have to place the same number of orders every day.) In Protosystem I, the design is based only on the time average and maximum number of records. The key predicate for QUANTITY-ORDERED-BY-STORE must be computable from the DSSL description. The appropriate information is contained in Store Ordering Generation and Store Order Generation.

Key predicates of output data sets of a computation are computed from the inputs and the definition of the computation. For example, consider Reorder Calculation, as shown in Figure 8.

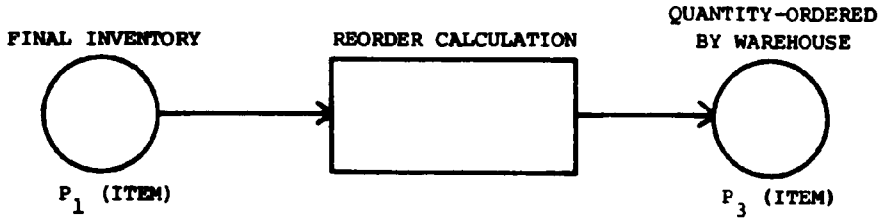


Figure 8

Suppose P (ITEM) is found to be just

ITEM ε SET-OF-ITEMS

From the definition of Reorder Calculation we have

P (ITEM) = P (ITEM) AND FINAL-INVENTORY

(ITEM) < 100.

Substituting in we get

P (ITEM) = ITEM ε SET-OF-ITEMS and

FINAL-INVENTORY(ITEM) < 100.

It is the job of the Question Answerer and Simulator to determine the time average number of records for which this predicate is true. The first term is independent of time; there are 4,000 items. Since all items have been defined to have the same behavior, the average can be found by finding the time average for which FINAL-INVENTORY(ITEM) < 100 is true for one item and multiplying by 4,000.

The above discussion should give the reader an idea of the level of description and the problems being attacked by the DSSL to DSL Translator, Question Answerer, and Simulator in Protosystem I. Let us turn to the interactive optimizer and heuristic optimizer.

Both optimizers play the same role; the interactive optimizer requires a human to suggest solutions; the heuristic optimizer is automatic. Only the interactive optimizer is debugged at this point. The problem faced by the optimizers in the A&T Micro case is shown schematically in Figure 9.

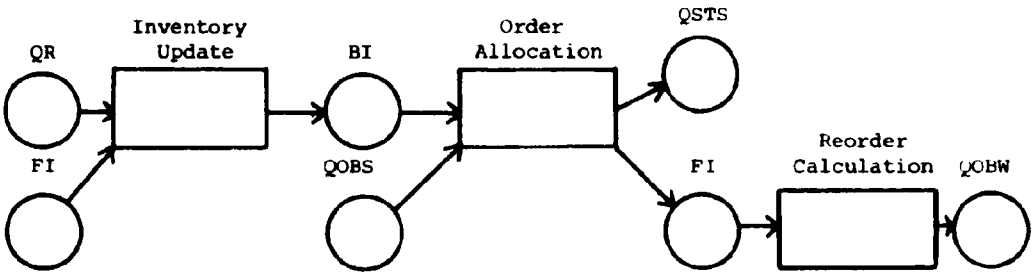


Figure 9

Circles represent data sets and rectangles represent computations.

Optimization involves merging computations and data sets, insertion of sorts, and assignment of file structure and parameters, and access methods. For example, in Figure 9 it would be possible to combine Inventory Update or Reorder Calculation into Order Allocation. If Inventory Update is combined with Order Allocation, data set BI and the associated file reads and writes can be eliminated. If Reorder Calculation is combined with Order Allocation a read of data set FI is eliminated, but FI cannot be eliminated, as it is an input to Inventory Update. The Order Allocation calculation requires that a running tabulation of the amount of each item remaining be kept while orders are being allocated to that item. This can either be done with a table in core containing an entry for each item, or data set QOBS can be sorted on item. If data set QOBS is sorted on item and calculation Order Allocation is done in item sequence, then data set BI can either be sorted on item and accessed sequentially or accessed randomly.

It should be clear that although we allow only three choices of file organization and a few access methods, that since the choices at any point partially constrain the adjacent choices the number of possible designs is combinatorially very large and not easy to enumerate.

The worth of a given design is found by estimating what it would cost to run it at the M.I.T. Computation Center IBM 370 installation. The center computes the cost of a run from a standard formula involving such variables as the amount of CPU time used, the amount of core used, and the number of secondary storage accesses made. The amount of core taken by a given run depends on such facts as how the operating system treats utility programs. Therefore, a large number of runs were made at that installation to get data on core requirements, sort times, and the like from which interpolations can be made for a proposed PL/I program or sort.

Once the above constraints have been chosen, PL/I and IBM Job Control Language can be generated by a process similar to code generation in a compiler. Thus, the output of the whole process is a set of PL/I programs for the given application.

MATHLAB GROUP

A. INTRODUCTION

This year the Mathlab Group has made significant progress on several fronts. Most notable has been the acceptance of the MACSYMA system (bugs and all) by a user community at M.I.T. and around the country. The system has already started to become a paper generating machine (published papers, that is). Significant progress has been made in algorithm analysis and design. Our new Greatest Common Divisor algorithm is exponentially faster than existing algorithms in many cases. Additional upgrading of our hardware and our LISP system has also taken place which has led to a significant improvement in the response time of the MACSYMA system. New capabilities (e.g. Laplace Transforms) have been added to the system and have in turn been critical in calculations which are now in various stages of publication. Our interaction with the Plasma Group at RLE has been yielding important results and we expect use of the system by plasma physicists in this country and possibly in Europe (via ARPA's connection to Norway). The MACSYMA system has also been made operational on the MULTICS system in the past year.

B. HARDWARE IMPROVEMENTS TO THE MATHLAB PDP-10

The "Mathlab" PDP-10 became operational in February, 1971. Very early it became clear that 256K of primary memory was insufficient to run more than one MACSYMA user at a time with a reasonable response time. Another 256K of memory was ordered from Ampex and 128K of it has been installed by April, 1973. The effect on swapping behavior of the system was dramatic, as expected. When the full 512K is installed, we expect five MACSYMAs to run simultaneously in memory. When additional improvements are made to our LISP system, this figure should double.

C. IMPROVEMENTS TO MAC-LISP

One of the most interesting improvements to our LISP system has been one which allows both users and designers to share the same code and obtain different debugging and run-time behavior. System designers want to be able to trace any subroutine at any given time. This is possible if all subroutine calls are run interpretively. This increases the run-time by a factor of about four on the average. Users do not wish to trace MACSYMA functions and thus should not pay this factor of four. We have made all subroutine calls go indirectly through special pages. One set of pages contains calls directly to the subroutine, the other set calls the subroutine linkage routine instead. By switching these pages in the user's page map we are able to get the performance desired. This work was done by our LISP development group made up of Jon White, Guy Steele, and Stavros Macrakis.

A normal MACSYMA version currently requires approximately 155K of memory, of which about 85K is shared code. A version containing the entire system would require about 225K, of which 135K would be shared. Approximately half of the unshared portion

of each user's space is comprised of list structure which is used as data by the subroutines. If we could share this data, then the memory overhead for additional MACSYMA users would decrease by a factor of two. We now plan to assign to each page in the user's map an entry in a table indicating the type of information in the pages (e.g. pure free storage, binary program, stack, impure free storage). This will give us some properties of segments (e.g. dynamic growth of number of pages containing a certain type of information) and still preserve the ability for one word to reference another directly. When this scheme is implemented the system will not only be able to share more information, but the arrangement of spaces for different information will change dynamically. Garbage collection and type testing times will also decrease sygnificantly.

D. ARPA NETWORK UTILIZATION

The Mathlab Machine went on the network in May, 1972. Since then we have built a moderate user community on the network. Our experience has been that once a user is able to solve a significant problem with the system then he is usually "hooked" and can be expected to use the system continually. A recent study indicates that in a period of several weeks 9% of all console hours charged to users came from network users, and that significant time was chargeable to users at 30 nodes along the network. Most of this network time is probably spent in using MACSYMA. Locally the machine is used quite heavily by the Medical Diagnosis and Automatic Programming groups as well as by the Mathlab Group and MACSYMA users in the M.I.T. community.

We are aware of significant projects at several sites. At NASA-Langley there is a project which uses MACSYMA to generate a finite element scheme for solving partial differential equations. FORTRAN subroutines are generated and fed to a CDC 6600 for the numerical computation [1]. Other computations in quantum electrodynamics are run there as well. At JPI MACSYMA is used by the numerical analysis group as an extension of their service to the JPL community. A Ph.D thesis in celestial astronomy is being completed, in part, through computations using MACSYMA. At Cal Tech calculations required in theoretical analyses of spline functions are being done and larger projects are under consideration. At the Stevens Institute of Technology calculations in plasma physics are being made. There is additional significant utilization of the system from the University of California at Santa Barbara [13], NOAA, and from within ARPA itself, but we are not familiar with the details of all of these applications. There is some discussion about utilization of the system from several plasma physics centers in Europe through the connection in Norway. These discussions are at a preliminary stage, however.

E. NEW AND IMPROVED SUBSYSTEMS IN MACSYMA

In the past year several new and some highly modified subsystems were introduced into the system and often led to success in calculations which were not possible earlier.

Richard Bogen completed Laplace Transform and Inverse Laplace Transform routines. We soon had an application in gas

chromotography from a professor at the Harvard School of Public Health which depended on this capability. The result apparently indicates the possibility of designing very general devices in this area.

Richard Zippel has completed a subsystem for manipulation formal power series with both negative and rational exponents. This system which is more general than similar work in other groups [8] adds an entirely new data representation to MACSYMA. It vindicates our general design decision to allow for a variety of data representations rather than depending on a single representation as is the case in most other systems. The subsystem has been used in many calculations.

Michael Genesereth has completed a translator from MACSYMA's top level language to LISP. There have been experiments by Richard Fateman which tied this translator to our recent LISP compiler and which show that for certain purely numerical calculations compiled MACSYMA routines run 20% faster than the corresponding FORTRAN programs compiled with the standard DEC compiler [7]. The reason for this discrepancy is due to the inefficiency in FORTRAN subroutine calls. This is ironic since LISP subroutine calls are more general because they allow for recursion and Standard FORTRAN does not.

David Yun has completed a version of our new EZ GCD algorithm which is also discussed below. This algorithm has been compared with the Modular GCD algorithm developed by Brown, Collins and Knuth [12]. As we had expected, the Modular algorithm required time which was an exponential function of its input size and the EZ algorithm only needed time which was a linear function of the input size on a class of problems. While the EZ algorithm is not exponentially better than previous algorithms for all problems, its advantage is sufficiently great to make those algorithms obsolete. In particular, certain calculations, especially polynomial factorizations, we had previously given up on have been successfully completed with the Ez GCD algorithm.

David Yun also has completed a subsystem for solving a set of polynomial equations [14]. This program uses a resultant algorithm to successively eliminate variables. It tries to keep the degrees of the resultants low by factoring them. A final univariate polynomial is solved, using infinite precisions arithmetic if necessary, to obtain its real roots to any desired degree of accuracy. The system can yield a surface of solutions when the original set of equations is undetermined. The system has been used to check calculations performed by Professor Rabin and Dr. Winograd of IBM.

Richard Bonneau of Project MAC's Theory Group has worked with Richard Fateman on Fast Fourier Transform algorithms for polynomial multiplications [6]. In certain cases when the polynomials are fairly dense, FFT techniques are more efficient than any other means for performing polynomial exponentiation as well as other processes. To our knowledge this is the first practical use of FFT techniques in algebraic manipulation.

F. WORK IN PROGRESS

Dr. Paul Wang has been extending his factorization system to factor polynomials over algebraic number fields. The mathematics (in particular, algebraic number theory techniques) has been developed by Elwyn Berlekamp, Hans Zassenhaus, and in our group by Linda Rothschild and by Peter Weinberger from the University of Michigan, who acted as a consultant. This system requires an extension of our rational function package to algebraic numbers. Barry Trager has been implementing that extension.

Trager has also been implementing an extension to our rational function package to allow it to handle polynomials in factored form. When this is completed we shall be able to avoid excessive blow-up of expressions in many situations.

Dr. Vera Pless has been considering implementing a system for calculations in group theory. This would require translating code in systems under development in Australia and Germany. She plans to extend such systems to handle different types of calculations in group theory, combinatorics, Lie groups, etc.

Joel Moses has been extending his implementations of the Risch integration algorithm [11]. The exponential case is almost completed now and work is proceeding on tools for handling the algebraic case. Jayant Shah of Northeastern has acted as a consultant on the algorithms from algebraic geometry required in this case. With Dr. Ed Ng of JPL, Moses has been analyzing the possibility of extending the Risch algorithm to a class of special functions, including the error function, the Incomplete Gamma function and the elliptic function.

Michael Genesereth has been implementing an extendable parser similar to that orginally devised by Vaughn Pratt. Preliminary experiments indicate that it is four times faster than our existing Floyd precedence parser. Many suggestions for extending our input syntax have been made and are being implemented.

Richard Bogen has written several versions of a MACSYMA Manual. The April, 1973 version is over 100 pages long, and is fairly complete [10]. Comments from users are likely to yield better versions of the manual and its accompanying Primer.

G. THE HENSEL LEMMA IN POLYNOMIAL MANIPULATION

Our major theoretical work has been in the design of the EZ GCD algorithm. The Hensel lemma used in this algorithm for extending the result from the univariate to multivariate case shows promise of applying in many other problems. David Yun has already used it to obtain efficient algorithms for square-free-factorizations and content calculations. He has also shown that a Hensel-like division algorithm is more efficient than the usual division algorithm on large polynomials. There is promise that this type of interpolation will yield a very efficient resultant algorithm. We believe that the use of the Hensel lemma is a breakthrough in this field which should lead to a class of algorithms which are the best possible ones or very close to the best for practically sized problems.

REFERENCES

1. Anderson, C.M., "Use of Symbolic Manipulations in the Development of Two-Dimensional Finite Elements", Book of Abstracts - SIAM 1973 National Meeting, Hampton, Virginia (June 1973).

2. Bers, A., Kulp, J., and Watson, D. C., "Analytic Description of Nonlinear Wave Interactions on a Computer", Bull. Amer. Phys. Soc., Series II, 17, No. 11, 991 (1972).

3. Bers, A., Kulp, J., and Watson, D. C., M.I.T.-R.L.E. Quarterly Progress Report No. 108, p. 167 (1973).

4. Bers, A., Kulp, J., and Watson, D. C., "Symbolic Computer Calculations of Plasma Wave Interactions", Book of Abstracts - International Congress on Waves and Instabilities, Institute of Theoretical Physics, Innsbruck, Austria, p. 13, (April 1973).

5. Bers, A., Karney, C. F. F., and Kulp, J., "Parametric Down Conversion from Lower-Hybrid Frequency Waves", Book of Abstracts - International Congress on Waves and Instability, Institute of Theoretical Physics, Innsbruck, Austria, p. 144 (April 1973).

6. Bonneau, Richard J., "A Class of Finite Computation Structures Supporting The Fast Fourier Transform", Book of Abstracts - SIAM 1973 National Meeting, Hampton, Virginia (June 1973).

7. Fateman, R. J., "Reply to an Editorial", SIGSAM Bulletin 25, March 1973, pp. 9-11.

8. Johnson, S. C., and Brown, W. S., "Truncated Power Series in ALTRAN", Book of Abstracts - SIAM 1973 National Meeting, Hampton, Virginia (June 1973).

9. Kulp, J. L., Bers, A., and Moses, J., "New Capabilities for Symbolic Computation in Plasma Physics", Book of Abstracts - Sixth Conference on Numerical Simulation of Plasmas, Lawrence Berkeley Laboratory, Berkeley, July 1973.

10. MACSYMA User's Manual, Version Four, April 1973, Project MAC, M.I.T.

11. Moses, J., "The Exponential Case of the Risch Integration Algorithm", (Invited Lecture), Book of Abstracts - SIAM 1973 National Meeting, Hampton, Virginia (June 1973).

12. Moses, J., and Yun, D. Y. Y., "The EZ GCD Algorithm", (to appear) Proceedings 1973 National ACM Conference, Atlanta, Georgia, August 1973.

13. Pickens, John R., ARPA Network Information Center, Doc. No. 16818, RFC No. 519, pp. 2-3.

REFERENCES (CON'T)

14. Yun, David Y. Y., "On Algorithms for Solving Systems of Polynomial Equations", Book of Abstracts - SIAM 1973 National Meeting, Hampton, Virginia (June 1973).

PLANNER

A. INTRODUCTION

Knowledge Based Programming is programming in an environment which has substantial knowledge of the semantic domain for which the programs are being written and of the purposes that the programs are supposed to satisfy. Actors are a semantic concept in which no agent is ever allowed to treat another as an object; instead a polite request must be extended to accomplish what the agent desires. Using actors the PLANNER Project is constructing a Programming Apprentice to make it easier for expert programmers to do knowledge based programming.

In the last year we have conceived and made a preliminary implementation of a modular ACTOR architecture and definitional method that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activations, or streams]. The architecture makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired in a uniform modular fashion. In fact it is impossible to determine whether a given actor is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

A satisfactory theory for the representation of knowledge should have one unified totally integrated formalism and semantics. For example we should not have one formalism and semantics for expressing declaratives and a separate formalism and semantics for expressing procedures. For some years now we have been working to achieve this goal. The record of our progress is published in the Proceedings of the International Joint Conferences on Artificial Intelligence beginning with the first conference in 1969. In the course of this research we have developed the Thesis of Procedural Embedding of Knowledge which is that "Knowledge of a domain is intrinsically bound up with the procedures for its use." An important corollary is that the fundamental technique of artificial intelligence is Automatic Programming and Procedural Knowledge Base Construction.

"Programs should not only work,
but they should appear to work as well."

PDP-1X Dogma

The PLANNER project is continuing research in natural and effective means for embedding knowledge in procedures. In the course of this work we have succeeded in unifying the

formalism around one fundamental concept: the ACTOR. Intuitively, an ACTOR is a potential performer which can play a role on cue. We use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. The ACTOR concept subsumes both the concept of data and the concept of instruction. The behavior of data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, grammars, demons, processes, contexts, and data bases can all be shown to be special cases of the behavior of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of these modes of behavior can be defined in terms of one kind of behavior: ACTOR TRANSMISSION. An actor is always invoked uniformly in exactly the same way regardless of whether it behaves as a recursive function, data structure, or process.

B. INTRINSIC COMPUTATION

We are approaching the problem of the representation of knowledge from a behavioral [procedural] as opposed to an axiomatic approach. Our view is that objects are defined by their actions rather than by axiomatizing the properties of the operations that can be performed on them.

"Ask not what you can do to some actor;
but what the actor can [will?] do for you."

Alan Kay has called this the INTRINSIC as opposed to the EXTRINSIC approach to defining objects. Our model follows the following two fundamental principles of organizing behavior:

Control flow and data flow are inseparable.

Computation should be done intrinsically instead of extrinsically i.e. "Every actor has the right to act for itself."

Although the fundamental principles are very general they have definite concrete consequences. For example they rule out the goto construct on the grounds that the goto violates the inseparability of control and data flow since the goto does not allow a message to be passed to the place where control is going. Also the goto defines a semantic object [the code following the tag] which is not properly syntactically delimited thus possibly leading to programs which are not properly syntactically nested. Similarly the classical interrupt mechanism of present day machines violates the principle of intrinsic computation since it wrenches control away from whatever instruction is running when the interrupt strikes.

"It is vain to multiply Entities beyond need."
William of Occam

"Monotheism is the Answer"

The unification and simplification of the formalisms for the procedural embedding of knowledge has a great many benefits for us:

FOUNDATIONS: The concept puts procedural semantics [the theory of how things operate] on a firmer basis. It will now be possible to do cleaner theoretical studies of the relation between procedural semantics and set-theoretic semantics such as model theories of the quantificational calculus and the lambda calculus.

LOGICAL CALCULI: A procedural semantics is developed for the quantificational calculi. The logical constants FOR-ALL, THERE-EXISTS, AND, OR, NOT, and IMPLIES are defined as actors.

PLANS are actors invoked by WORLD DIRECTED INVOCATION [invocation on the basis of a fragment of a micro-world] to try to achieve some goal. PROCEDURAL DATA BASES [WORLDS] are actors which organize a set of actors for efficient retrieval. There are three primitive operations for data bases: PUT, GET, and ERASE which are done on the basis of world directed invocation which the worlds do at the behest of the plans that they serve.

KNOWLEDGE BASED PROGRAMMING is programming in an environment which has a substantial knowledge base in the application area for which the programs are intended. The actor formalism aids knowledge based programming in the following ways:

PROCEDURAL EMBEDDING of KNOWLEDGE

TRACING BEHAVIORAL DEPENDENCIES

SUBSTANTIATING that ACTORS SATISFY their INTENTIONS

INTENTIONS: Furthermore the confirmation of properties of procedures is made easier and more uniform. Every actor has an INTENTION which checks that the prerequisites and the context of the actor being sent the message are satisfied. The intention is the CONTRACT that the actor has with the outside world. How an actor fulfills its contract is its own business. By a SIMPLE BUG we mean an actor which does not satisfy its intention. We would like to eliminate simple debugging of actors by the META-EVALUATION of actors to show that they satisfy their intentions. The rules of deduction to establish that actors satisfy their intentions essentially take the form of a high level interpreter for abstractly evaluating the program in the context of its intentions. This process [called META-EVALUATION] can be justified by a form of induction. Meta-evaluation captures a large part of the mechanism that a programmer goes thru when he reads a piece of code to determine that it will satisfy its intended purpose. In general in order to substantiate a property of the behavior of an actor system some form of induction will be needed. At present, actor induction for an actor configuration with audience E can be tentatively described in the following manner:

1. The actors in the audience E satisfy the intentions of the actors to which they send messages

and

2. For each actor A [including those created in the course of a computation] the intention of A is satisfied => the intentions for all actors sent messages by A are satisfied

Therefore

The intentions of all actions caused by E are satisfied (i.e. the system behaves correctly)

Computational induction [Manna], structural induction [Burstall], and Peano induction are all special cases of ACTOR induction. Actor based intentions have the following advantages over previous formalisms that have been proposed:

The intention is decoupled from the actors it describes.

We can partially substantiate facts about the behavior of actors without giving a complete formal proof. An actor who is asked can if it chooses vouch for some circumstance being the case. At some later time if we require further justification, then we can re-examine the situation.

Intentions of concurrent actions are more easily disentangled.

We can more elegantly write intentions for dialogues between actors.

The intentions are written in the same formalism as the procedures they describe. Thus intentions can have intentions. Furthermore intentions for side effects are expressible without recourse to the notion of a global state. The extent to which intentions are checked at execution time as opposed to being verified once and for all (making the execution time check superfluous) becomes at least partially an economic decision. Sometimes [as in type checking] it is cheaper to use an efficient runtime check providing that the possibility of a run time fault is tolerable.

Because protection is an intrinsic property of actors, we hope to be able to deal with protection issues in the same straightforward manner as more conventional intentions.

Intentions of data structures are handled by the same machinery as for all other actors.

The flow chart inductive assertion method of Floyd, the axiomatic rules for PASCAL of Hoare, and their extension

to SIMULA-67 style processes by Clint are all special cases of meta-evaluation.

COMPARATIVE SCHEMATOLOGY: The theory of comparative power of control structures is extended and unified. The following hierarchy of control structures can be explicated by incrementally increasing the power of the actor transmission primitive.

iterative-->recursive-->backtrack-->determinate-->universal

EDUCATION: The model is sufficiently natural and simple that it can be made the conceptual basis of the model of computation for students. In particular it can be used as the conceptual model for a generalization of Seymour Papert's "little man" model of LOGO. The model becomes a cooperating society of "little men" each of whom can address others with whom it is acquainted and politely request that some task be performed.

LEARNING AND MODULARITY: Actors also enable us to teach computers more easily because they make it possible to incrementally add knowledge to procedures without having to rewrite all the knowledge which the computer already possesses. Incremental extensions can be incorporated and interfaced in a natural flexible manner. Protocol abstraction [abstracting general procedures from the protocols of their execution on particular cases: Hewitt 1969, 1971; Hart, Nilsson, and Fikes 1972; Sussman 1972] can be generalized to actors so that procedures with an arbitrary control structure can be abstracted.

EXTENDABILITY: The model provides for only one extension mechanism: creating new actors. However, this mechanism is sufficient to obtain any semantic extension that might be desired.

PRIVACY AND PROTECTION: Actors enable us to define effective and efficient protection schemes. Ordinary protection falls out as an efficient intrinsic property of the model. The protection is based on the concept of "use". Actors can be freely passed out since they will work only for actors which have the authority to use them. Mutually suspicious "memoryless" subsystems are easily and efficiently implemented. ACTORS are at least as powerful a protection mechanism as domains [Schroeder, Needham, etc.], access control lists [MULTICS], objects [Wulf 1972] and capabilities [Dennis, Plummer, Lampson]. Because actors are locally computationally universal and cannot be coerced there is reason to believe that they are a universal protection mechanism in the sense that any other protection mechanism can be efficiently defined using actors. The most important issues in privacy and protection that remain unsolved are those involving intent and trust. Here the concept of justification plays an important role. A protected subsystem that provides an answer should be able to justify that the answer is correct. We are currently considering ways in which our model can be further developed to address this problem.

SYNCHRONIZATION: Serializers provide at least as powerful a synchronization mechanism as semaphores with no busy waiting and guaranteed first in first out discipline on each resource. A synchronization actor is easier to use and substantiate than a semaphore [Dijkstra 1971] since they are directly tied to the control-data flow. Also it provides more protection because no activator [agent process] can get thru the serializer until the current guard for the serializer is given the go ahead and a new guard is provided for each activator that goes thru the serializer.

SIMULTANEOUS GOALS: The synchronization problem is actually a special case of the simultaneous goal problem. Each resource which is seized is the achievement and maintenance of one of a number of simultaneous goals. Recently Sussman has extended the previous theory of goal protection by making the protection guardians into a list of predicates which must be evaluated every time anything changes. We have generalized protection in our model by endowing each actor with a scheduler and an intention. We thus retain the advantages of local intentional semantics. A scheduler actor allows us to program EXCUSES for violation in case of need and to allow NEGOTIATION and re-negotiation between the actor which seeks to seize another and its scheduler. Richard Waldinger has pointed out that the task of sorting three numbers is a very elegant simple example illustrating the utility of incorporating these kinds of excuses for violating protection.

RESOURCE ALLOCATION: Each actor has a banker who can keep track of the resources used by the actors that are financed by the banker.

STRUCTURING: The actor point of view raises some interesting questions concerning the structure of programming.

> STRUCTURED PROGRAMS: We maintain that actor communication is well-structured. Having no goto, interrupt, semphore, or other constructs, they do not violate "the letter of the law". Some readers will probably feel that some actors exhibit "undisciplined" control flow. These distinctions can be formalized through the mathematical discipline of comparative schematology [Patterson and Hewitt].
>
> STRUCTURED PROGRAMMING: Some authors have advocated top down programming. We find that our own programming style can be more accurately described as "middle out". We typically start with specifications for a large task which we would like to program. We refine these specifications attempting to create a program as rapidly as possible. This initial attempt to meet the specifications has the effect of causing us to change the specificatons in two ways:
>
> > 1: More specifications [features which we originally did not realize are important] are added to the definition of the task.

2: The specifications are generalized, specialized, and otherwise combined to produce a task that is easier to implement and more suited to our real needs.

IMPLEMENTATION: Actors provide a very flexible implementation language. In fact we are carrying out the implementation entirely in the formalism itself. By so doing we obtain an implementation that is efficient and has an effective model of itself. The efficiency is gained by not having to incur the interpretive overhead of embedding the implementation in some other formalism. The model enables the formalism to answer questions about itself and to draw conclusions as to the impact of proposed changes in the implementation.

ARCHITECTURE: Actors can be made the basis of the architecture of a computer which means that all the benefits listed above can be enforced and made efficient. Programs written for the machine are guaranteed to be syntactically properly nested. The basic unit of execution on an actor machine is sending a message much in the same way that the basic unit of execution on present day machines is an instruction. On a current generation machine in order to do an addition an add instruction must be executed; so on an actor machine a hardware actor must be sent the operands to be added. There are no goto, semaphore, interrupt, or other instructions on an ACTOR machine. An ACTOR machine can be built using the current hardware technology that is competitive with current generation machines.

"Now! Now!" cried the Queen.
"Faster! Faster!"
Lewis Carroll

Current developments in hardware technology are making it economically attractive to run many physical processors in parallel. This leads to a "swarm of bees" style of programming. The actor formalism provides a coherent method for organizing and controlling all these processors. One way to build an ACTOR machine is to put each actor on a chip and build a decoding network so that each actor chip can address all the others. In certain applications parallel processing can greatly speed up the processing. For example with sufficient parallelism, garbage collection can be done in a time which is proportional to the logarithm of the storage collected (instead of a time proportional to the amount of storage collected which is the best that a serial processor can do). Also the architecture looks very promising for parallel processing in the lower levels of computer audio and visual processing.

"All the world's a stage,
And all the men and women merely actors.
They have their exits and their entrances;
And one man in his time plays many parts."

"If it waddles like a duck, quacks like a duck, and otherwise behaves like a duck; then you can't tell that it isn't a duck."

C. ADDING AND REORGANIZING KNOWLEDGE

Our aim is to build a firm procedural foundation for problem-solving. The foundation attempts to be a matrix in which real world problem solving knowledge can be efficiently and naturally embedded. In short the problem is to "get the knowledge to where the action is." We envisage knowledge being embedded in a set of knowledge boxes with interfaces between the boxes. In constructing models we need the ability to embed more knowledge in the model without having to totally rewrite it. Certain kinds of additions can be easily encompassed by declarative formalisms such as the quantificational calculus by simply adding more axioms. Imperative formalisms such as actors do not automatically extend so easily. However, we are implementing mechanisms that allow a great deal of flexibilty in adding new procedural knowledge. The mechanisms attempt to provide the following abilities:

PROCEDURAL EMBEDDING: They provide the means by which knowledge can easily and naturally be embedded in processes so that it will be used as intended.

CONSERVATIVE EXTENSION: They enable new knowledge boxes to be added and interfaced without rewriting all the previous knowledge.

MODULAR CONNECTIVITY: They make it possible to reorganize the interfaces between knowledge boxes.

MODULAR EQUIVALENCE: They guarantee that any box can be replaced by one which satisfies the previous interfaces.

Actors must provide interfaces so that the binding of interfaces between boxes can be controlled by knowledge of the domain of the problem. The right kind of interface promotes modularity because the procedures on the other side of the interface are not affected so long as the conventions of the interface are not changed. These interfaces aid in debugging since traps and checkpoints are conveniently placed there. More generally, formal conditions can be stated for the interfaces and confirmed once and for all.

D. UNIFICATION

We claim that there is a common intellectual core to the following (now somewhat isolated) fields that can be characterized and investigated:

digital circuit designers
data base designers
computer architecture designers
programming language designers

computer system architects

> "Our primary thesis is that there can and must exist a single language for software engineering which is usable at all stages of design from the initial conception through to the final stage in which the last bit is solidly in place on some hardware computing system."
>
> Doug Ross

The time has come for the unification and integration of the facilities provided by the above designers into an intellectually coherent manageable whole. Current systems which separate the following intellectual capabilities with arbitrary boundaries are now obsolete.

"Know thyself".

We intend that our system should have a useful working knowledge of itself. That is, it should be able to answer reasonable questions about itself and be able to trace the implications of proposed changes in itself.

> "We base ourselves on the idea that in order for a program to be capable of learning something it must first be capable of being told it. In fact, in the early versions we shall concentrate entirely on this point and attempt to achieve a system which can be told to make a specific improvement in its behavior with no more knowledge of its internal structure or previous knowledge than is required in order to instruct a human."
>
> John McCarthy 1958

Representing in a usable way the knowledge about how a problem solver works is the first step towards teaching it how to do new things instead of always telling it how to do them at a very low level. Also it is the only way in which the problem solver can have anything but the most superficial understanding of its own behavior. The implementation of actors on a conventional computer is a relatively large complex useful program which is not a toy. The implementation must adapt itself to a relatively unfavorable environment. It illustrates the techniques and difficulties of large software systems. Creating a model of itself should aid in showing how to create useful models of other large knowlege based programs since the implementation addresses a large number of difficult semantic issues. We have a number of experts on the domain that are very interested in formalizing and extending their knowledge. These experts are good programmers and have the time, motivation, and ability to embed their knowledge and intentions in the formalism.

"The road to hell is paved with good intentions."

Once the experts put in some of their intentions they find that they have to put in a great deal more to convince the

auditor of the consistency of their intentions and procedures. In this way we hope to make explicit all the behavioral assumptions that our implementation is relying upon. The domain is closed in the sense that the questions that can reasonably be asked do not lead to a vast body of other knowledge which would have to be formalized as well. The domain is limited in that it is possible to start with a small superficial model of actors and build up incrementally. Any advance is immediately useful in aiding and motivating future advances. There is no hidden knowledge as the formalism is being entirely implemented in itself. The task is not complicated by unnecessary bad software, engineering practices such as the use of gotos, interrupts, or semaphores.

E. HIERARCHIES

The model provides for the following orthogonal hierarchies:

SCHEDULING: Every actor has a scheduler which determines when the actor actually acts after it is sent a message. The scheduler handles problems of synchronization. Another job of the scheduler [Rulifson] is to try to cause actors to act in an order such that their intentions will be satisfied.

INTENTIONS: Every actor has an intention which makes certain that the prerequisites and context of the actor being sent the message are satisfied. Intentions provide a certain amount of redundancy in the specification of what is supposed to happen.

MONITORING: Every actor can have monitors which look over each message sent to the actor.

RESOURCE MANAGEMENT: Every actor has a banker which monitors the use of space and time.

Note that every actor has all of the above abilities and that each is done via an actor!

> "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"
>
> Lewis Carroll

The previous sentence may worry the reader a bit as she [he] might envisage an infinite chain of actions [such as banking] to be necessary in order to get anything done. We short circuit this by only requiring that it appear that each of the above activities is done each time an actor is sent a message.

"There's no use trying," she said: "one can't believe impossible things."
"I daresay you haven't had much practice," said the Queen. "When I was your age, I always did it for half-an-hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."
Lewis Carroll

Each of the activities is locally defined and executed at the point of invocation. This allows the maximum possible degree of parallelism. Our model contrasts strongly with extrinsic quantificational calculus models which are forced into global noneffective statements in order to characterize the semantics.

"Global state considered harmful."

We consider language definition techniques [such as those used with the Vienna Definition Language] that require the semantics be defined in terms of the global computational state to be harmful. Formal penalties [such as the frame problem and the definition of simultaneity] must be paid even if the definition only effectively modifies local parts of the state. Local intrinsic models are better suited for our purposes.

F. SYNTACTIC SUGAR

"What's the good of Mercator's North Poles and Equators,
Tropics, Zones and Meridian Lines?"
So the Bellman would cry: and the crew would reply
"They are merely conventional signs!"
Lewis Carroll

Thus far in our discussion we have discussed the semantic issues intuitively but vaguely. We would now like to proceed with more precision. Unfortunately in order to do this it seems necessary to introduce a formal language. The precise nature of this language is relatively unimportant so long as it is capable of expressing the semantic meanings we wish to convey.

"Use throw away implementations."
Alan Kay

"But make each one good enough to tell you what you need to know to make the next!"
Tony Hoare

For some years we have been constructing a series of languages to express our evolving understanding of the above semantic issues. The latest of these is called PLANNER-73.

"Is it garbage yet?"

Meta-syntactic variables will be underlined. We shall assume that the reader is familiar with advanced pattern matching languages such as SNOBOL4, CONVERT, QA4, and PLANNER-71.

Consider the problem of adding 2 to x where x denotes 3 We will have an actor 2 which given + and x will send us back the sum. It may seem somewhat strange to have 2 as an actor but this is the point of view taken by Alonzo Church in his paper on the lambda calculus and some string processing interpretive languages operate in this way. Also the SMALL TALK language of Alan Kay has taken up this view and shown how it can be used to systematize type coercions. We will denote sending 2 the + and x by (2 + x). We want to send the actor 2 only one message in order to accomplish the addition. So we shall agree that (2 + x) really means (%2 [+ x]%) where [+ x] is a tuple whose first element is + and whose second element is x. The actor 2 will need to be able to convert x into its denotation 3 so we shall need to send it an environment E with x=3 to tell it how to do so. Thus the sum can be further analyzed to be

```
(=> (#eval =E)
    (%%2
        (#by-expression
            [+ x]
            (#environment E))%%)).
```

Reflecting on the message sent 2, we realize that the actor 2 needs to be told a continuation C to send the answer when it finishes and so we agree that the sum can be further analyzed as

```
(=>>> (#eval =E (#continuation =C))
    (%%%2
        (#transmission
            (#by-expression
                [+ x]
                E)
            (#continuation C))%%%))
```

At this point we shall not carry our analysis of the sum any further but instead shall reflect on what we have done. We shall use [s1 s2 ... sn] to denote the finite sequence s1, s2, ... sn. A sequence s is an actor where (s i) is element i of the sequence s. For example ([a (2 + 3) b] 2) will send "back" 5. We will allow the possibility that the expressions enclosed between "[" and "]" may be evaluated concurrently. We use "(" and ")" to denote the simultaneous synchronous transmission of a sequence of messages so that (A1 A2 ... An) will be defined to be (%A1 [A2 ... An]%). The sequence expression [%a1 a2 ... an%] (read as "a1 then a2 ... finally send back an") will be evaluated by evaluating a1, a2, ..., and an in sequence and then sending back ["returning"] the value of an as the message.

Identifiers can be created by the prefix operator =. For example if the pattern =x is matched with v, then a new identifier is created and bound to v.

"But 'glory' doesn't mean 'a nice knock-down argument,'" Alice objected.

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean--neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master--that's all."

Lewis Carroll

Humpty Dumpty propounds two criteria on the rules for names:

Each actor has complete control over the names he uses.

All other actors must respect the meaning that an actor has chosen for a name.

We are encouraged to note that in addition to satisfying the criteria of Humpty Dumpty, our names also satisfy those subsequently proposed by Bill Wulf and Mary Shaw:

The default is not necessarily to extend the scope of a name to any other actor.

The right to access a name is by mutual agreement between the creating actor and each accessing actor.

An access right to an actor and one of its acquaintances is decoupled.

It is possible to distinguish different types of access.

The definition of a name, access to a name, and allocation of storage are decoupled.

The use of the prefix = does not necessarily imply the allocation of any storage.

One of the simplest kinds of ACTORS is a cell. A cell with initial contents V can be created by evaluating (cons-cell V). Given a cell x, we can ask it to send back its contents by evaluating (contents x) which is an abbreviation for (%x (#contents)%). For example (contents (cons-cell 3)) evaluates to 3. We can ask it to change its contents to v by evaluating (x <- v). For example if we let x be (cons-cell 3) and evaluate (x <- 4), we will subsequently find that (contents x) will evaluate to 4.

The pattern (by-reference P) matches object E if the pattern P matches (cons-cell E), i.e. a "cell" [see below] which contains E. Thus matching the pattern (by-reference =x) against E is the same as binding x to (cons-cell E), i.e. a new cell which contains the value of the expression E. We shall use => [read as "RECEIVE MESSAGE"] to mean an actor which is reminiscent of the actor LAMBDA in the lambda calculus. For example (=> =x body) is like (LAMBDA x body) where x is an identifier. An expression (=> pattern body) is an abbreviation for

```
(===>
    (#transmission
        pattern
        (#continuation =c))
    (%%%c
        (#transmission body
            (#continuation none))%%%))
```

where ===> is a more general actor that, unlike =>, does not implicitly bind the continuation.
Evaluating

(%(=> pattern body) the-message%), i.e. sending the-message to
(=> pattern body),

will attempt to match the-message against pattern. If the-message is not of the form specified by pattern, then the actor is NOT-APPLICABLE to the-message. If the-message matches pattern, then body is evaluated.

Evaluating (%(cases [f1 f2 ... fn]) arg%) will send f1 the message arg and if it is not applicable then it will send f2 the message arg, etc. until it finds one that is applicable. The message (#not-applicable) is sent back to the complaint-dept if none were applicable. Evaluating (%(cases [f1 f2 ... fn]) arg%) will send f1 the message arg, ..., and send fn the message arg concurrently.

Abbreviations

The following abbreviations will be used to improve readability:

(rules object clauses) for

(%(cases clauses) object%)

```
(let
    {
        [x0 = expression0]
        [x1 = expression1]
        ...
        [xn = expressionn]}
    body) for

((=> [=x0 =x1 ... =xn] body)
    expression0
    expression1
     ...
    expressionn)
```

G. ACTOR TRANSMISSION

The world's a theatre, the earth a stage,
Which God and nature do with actors fill.
Thomas Heywood 1612

Consider the event of transmitting an M to a target T.

(%%%T M%%%)

If the target T is the following:

```
(===>
   the-pattern-for-the-transmission
   the-body)
```

then the-body is evaluated in an environment where the-pattern for-the-transmission is bound to M.

Suppose that we have a TROUPE of actors Tr with a distinguished subset C known as the CAST. The cast C is distinguished by being able to directly interact immediately with the audience. We define an EVENT to be a quadruple of the form [T M A EC] where T is the target, M is the message, A is the activator [agent, process] propelling the message, and EC is the event count of the activator A. We define a HISTORY to be a strict partial order of events with the transitive closure of the partial ordering -> [read as PRECEDES] where

[t1 m1 a1 ec1] -> [t2 m2 a2 ec2] if

a1 = a2 and ec1 < ec2

or

m1 is created in the action [t1 m1 a1 ec1]
and {m1} intersect {t2 m2} is nonempty.

The above definiton states that one event precedes another if they have the same activator and the event count of one is less than the event count of the other. A history will be said to be WELL-FORMED if any member of the troupe which is not a member of the cast is first sent to a member of the audience before being the target of the first event of an EXOGENOUS activator. The intent of this restriction is to prevent the audience from arbitrarily affecting the internal workings of the troupe and to further clarify the nature of the boundary between the troupe and its external environment as represented by the audience. We will assume that all our histories are well-formed. e allow the range of event counts for an activator [agent, process] to be a [possibly infinite] segment of the integers [including negative integers]. The definition can be generalized to cover events which have different activators by analyzing how activators are created and absorbed. The relation -> can be thought of as the "arrow of time" which we require to be a strict partial order. That is, there is no event e such that e->e is the case.

The constraints on partial orders given below have been extracted from a forthcoming paper "Behavioral Semantics of Actor Systems" by Irene Greif and Carl Hewitt. Every history involving a cell c satisfies the following constraints:

Guaranteed Reply

Suppose
An event E of the form [c (#transmission (#contents) (#continuation r)) A EC] of of the form [c (#transmission [<- x r] (#continuation r)) A EC] is in the history.
Then
E -> [r ? A EC'] is in the history.

Retrieve the last contents stored.

Suppose
Let E1 be of the form [c (#transmission [<- x] (#continuation r1)) ? ?] and E2 be of the form [c (#transmission (#contents) (#continuation r2)) A EC]. If E1 -> E2 is in the history and for every event E in the history such that E is of the form [c (#transmission [<- y] ?) ? ?] then E -> E1 or E2 -> E.
Then
E2 -> [r2 x A EC'] is in the history.

Certain primitive actors such as SERIALIZERS (which only let one message thru at a time) impose additional constraints on the partial order. Every history involving a serializer s satisfies the following constraints:

No butting in front.

Suppose
two events of the following form are in the history:
[s (#thru r1) A1 EC1] -> [s (#thru r2) A2 EC2]

Then
if E2 = [r2 (#guard g2) A2 EC2'] is an event in the history then there are events E1 and E3 in the history such that
E1 = [r1 (#guard g1) A1 EC1']
E3 = [g1 (#unlock) ? ?]
E1 -> E3 -> E2

Guaranteed reply provided no activator locks the serializer forever.

Suppose
E1 = [s (#thru r1) A1 EC1] is an event in the history and that for every event E of the form [s (#thru r) A EC] either E1 --> E or there are events [r (#guard g) A EC'] and [g (#unlock) ? ?] in the history.
Then
there is an event [r1 (#guard g1) A1 EC1'] in the history.

Notice that we do not require a definiton of global simultaneity; i.e. we do not require that two arbitrary events be related by ->. An event E1 can CAUSALLY AFFECT an event E2 only if E1->E2. We can draw a fixed but otherwise arbitrary boundary around a troupe of actors TP in order to study their behavior with respect to an external configuration of actors [called the AUDIENCE]. Where the boundary is drawn will depend on the reason for attempting to isolate the behavior of the troupe. We define the BEHAVIOR of a history with respect to an audience to be the subpartial ordering of the history consisting of those quadruples [T M A EC] where the target T is an element of the audience or the quadruple is the first event of an EXOGENOUS activator. The REPERTOIRE of the troupe TP is the class of all the behaviors of TP with respect to all audiences for the fixed external boundary. The REPERTOIRE of a troupe defines what the troupe can do as opposed to how it performs. Two actors will be said to be EQUIVALENT if they have the same REPERTOIRE. For example (cons-cell 5) is equivalent to ((cons-cell 3) <- 5).

We can name an actor defined by D with the name N in the body B by the notation (labels {[N <= D]} B). More precisely, the behavior of the actor (labels {[f <= (E f)]} B) in the body B is defined by the MINIMAL BEHAVIORAL FIXED POINT of (E f) i.e. the minimal repertoire M such that (E M) = M. In the case where M happens to define a function, it will be the case that the repertoire M is isomorphic with the graph [set of ordered pairs] of the function defined by M and that the graph of M is also the least (lattice-theoretic) fixed point of Park and Scott. We shall use (<= N D) as an abbreviation for

(label {[N <= D]} N). For example

```
(
    (<=
        factorial
        (cases
            [(=> [0] 1)
            (=> [=n] (n * (factorial (n - 1))))]))
    3) evaluates to 6
```

H. SIDE EFFECTS

It is sometimes necessary to be able to preserve the complete history of events. The only events in our model occur when an actor is sent a call. Intuitively a SIDE EFFECT has occurred if there is some question for an actor which has a different answer when asked of the actor on two different occasions. A side effect can be localized in space-time around an event E by the following mechanism:

> There is an actor T1 and message M1 such that IF the event E1= [T1 M1 A1 EC1] [where as before A1 is an activator and EC1 is the event count for the activator] were to happen before E, then some later event for activator A1 would have a different transmission than IF E1 happened after E.

Side effects destroy information. Therefore the actor transmission primitive must not in itself necessarily have side effects. The side effects in our model stem from other actors with side effects; they are not derived from the actor transmission primitive.

I. MANY HAPPY RETURNS

Many actors who are executing in parallel can share the same continuation. They can all send a message ["return"] to the same continuation. This property of actors is heavily exploited in meta-evaluation and synchronization. An actor can be thought of as a kind of virtual processor that is never "busy" [in the sense that it cannot be sent a message].

The basic mechanism of sending a message preserves all relevant information and is entirely free of side effects. Hence it is most suitable for purposes of semantic definition of special cases of invocation and for debugging situations where more information needs to be preserved. However, if fast write-once optical memories are developed then it would be suitable to be implemented directly in hardware.

The following is an overview of what appears to be the behavior of the process of an activator A transmitting a message M to a target T.

1: Call the banker of A to approve the expenditure of resources by the caller.

2: The banker will probably eventually send a message to the scheduler of T.

3: The scheduler will probably eventually send a message to the monitor manager of T.

4: The monitor manager will probably eventually send a message to the intention manager of T.

5: The intention manager of T will probably eventually send the message M to T.

6: The activator A will finally attempt to get some real work done by doing T's thing.

There are several important things to know about the process of sending a message to an actor:

1: Actor transmission is a universal control primitive in the sense that control operations such as function calls, iteration, coroutine invocations, resource seizures, scheduling, synchronization, and continuous evaluation of expressions are special cases.

2: Actors can conduct their dialogue directly with each other; they do not have to set up some intermediary such as ports [Krutar, Balzer, and Mitchell] or possibility lists [McDermott and Sussman] which act as pipes through which conversations must be conducted.

3: Actor transmission is entirely free of side effects [such as those in the message mechanism of the current SMALL TALK machine of Alan Kay, in the port mechanism of Kruter and Balzer, and in possibility lists of McDermott and Sussman] Being free of side effects allows us a maximum of parallelism and allows an actor to be engaged in several conversations at the same time without becoming confused.

4: Actor transmission makes no presupposition that the actor sent the message will ever send "back" a message or that "back" is even defined. The unidirectional nature of sending messages enables us to define iteration, monitors, coroutines, etc. straightforwardly.

5: The ACTOR model is not an [environment-pointer, instruction-pointer] model such as the CONTOUR model. A continuation is a full blown actor [with all the rights and privileges]; it is not a program counter. There are no instructions [in the sense of present day machines] in our model. Instead of instructions, an actor machine has certain primitive actors built in hardware.

6: All of the control constructs listed below are universal in some sense; but the actor transmission primitive in not an immediate special case of any one of them.

6.1 GOTO is not as general because it does not allow a message to be sent to the target.

6.2 FUNCTION CALL is too specialized because it always binds a continuation to the sender.

6.3 SIMULA-67 CLASS INSTANTIATION is too specialized because a class cannot be further instantiated and because control must return to the instantiator immediately after a detach of the class instance. The only other way out of the class is to use a RESUME statement. RESUME [the SIMULA-67 coroutine primitive] is separate from class instantiation and unfortunately does not allow a message to be passed to the process being resumed without a gratuitous side-effect.

6.4 Actor transmission at this level is very simular to the objects constructed by the J operator of Peter Landin and the ESCAPE construct of John Reynolds. The major contribution here is the observation that the function call can be defined as a special case. Major differences show up at the next lower level where the protocol with the underlying activator [agent process] is made explicit.

All of the above control constructs are trivially special cases of actor transmission.

J. DATA BASES

Data bases are actors that organize a set of actors for efficient retrieval. There are three primitive operations on data bases: PUT, GET, and ERASE. A new virgin default data base can be created by evaluating (virgin). If we let W = (virgin), then W will be a virgin world. We can put an actor (at John airport) in the world W by evaluating (use-world W (put (at John airport))). We could add further knowledge by evaluating

(use-world W (put (at airport Boston))) to record that the airport is at Boston.

(use-world W (put (city Boston))) to record that Boston is a city.

If the constructor EXTENSION is passed a world w then it will create a new default world which is an extension of w. For example

```
(let {[W' = (extension W)]}
    (use-world W'
        (put
            (on John (flight 34)))))
```

will bind W' to a new world in which we have supposed that John is on flight #34. The world W is unaffected by this operation. On the other hand the extension world is affected if we do (use-world W (put (hungry John))).

Worlds can ask the actors put in them to index themselves for rapid retrieval. Simple retrieval can be done using patterns. For example:

```
(get (at ? ?) (#then Receiver) (#else Alternative))
```

puts Receiver in an environment to retrieve all the actors in W which match the pattern (at ? ?). Now GET will thus retrieve either (at airport Boston) or (at John airport). We do not want to have to explicitly store every piece of knowledge which we have but would like to be able to derive conclusions from what is already known: We can distinguish several different classes of procedures for deriving conclusions. The actor >=> [read as "ON TRIGGER"] with the syntax

(>=> pattern-for-trigger body)

creates a PLAN that can be invoked by pattern directed invocation by a trigger which matches pattern-for-trigger.

K. PATTERN DIRECTED INVOCATION

Plans communicate thru making assertions, erasures, and denials using the world machinery. We assume the existence of a generator ANONYMOUS which generates new anonymous individuals anon1, anon2, etc. which have never before been encountered. To show the utility of such a generator consider the problem of proving (subset x z) [x is a subset of z] where we have a world which contains:

```
(subset x w)

(subset x y)

(subset y z)

(subset (union x s) t)

[prove-subset <=
    (>=> (prove (subset =a =c))
        (prove (subset a =b)
            (#then
                (prove (subset b c))))))]

[prove-subset-union <=
    (>=> (prove (subset =a (union =a ?)))
        (done))]
```

The problem is solved by "wishful thinking." In order to find b such that (subset x b) we let b be an anon1 which is a never before encountered individual which we wish to have certain properties. Then we note that anon1 might be w. But we are unable to prove (subset w z) so we reconsider and see that anon1 might be y. We successfully prove (subset y z) and so the problem is solved.

Now consider the problem of proving (subset x t). As above let b be anon2 to try to satisfy (subset x b). We find that neither w nor y work out as anon1 so we try the plan prove-subset-union. Thus it is sufficient that anon2 be (union x ?) where we don't know what ? is yet. We hopefully continue trying to show that (subset anon2 t) and find that we would be done if only anon2 were (subset x s). This is satisfactory if we let ? be s and so we have solved the problem.

L. McCARTHY AND THE AIRPORT

We would like to illustrate some uses for statements about the possibility of McCarthy being at the airport to illustrate the point that what counts is not whether some particular statement is TRUE or FALSE but rather the uses to which the statement can be put.

```
McCarthy is at the airport.
(put (at McCarthy airport))
```

```
If a person is at the airport, then the person might take a
plane from the airport.
   [put-at <=
        (>=> (put (at =person airport))
             (put (might (take-plane-from person airport))))]

McCarthy is not at the airport.
(deny (at McCarthy airport))

If a person is not at the airport then he can't take a plane
from the airport.
    [deny-at <=
        (>=> (deny (at =person airport))
             (put (can't (take plane from person airport))))]

It is not known whether McCarthy is as the airport.
(erase (at McCarthy airport))

If it is not known whether a person is at the airport then
erase whatever depends on previous knowledge that the person
is at the airport.
    [erase-at <=
        (>=> (erase (at =person airport))
                (find-all (depends-on =s (at person airport))
                        (#then (erase s))))]

Get McCarthy to the airport.
(achieve {(at McCarthy airport)})

To achieve a person at a place:

     Find the present location of the person.
     Show that it is walkable from the present location to the
car.
     Show that is drivable from the car to the place.

    [achieve-at <=
        (>=> (achieve (at =person =place))
             (achieve
                 (find (at person =present-location)
                     (#then (show (walkable present-location
                                            car)
                          (#then
                              (show (drivable car place)
                                  (#else (make-plan (at person
                                                        place))))))))))]

Show that McCarthy is at the airport.
(show (at McCarthy airport))
```

To show that a thing is at a place show that the thing is at some intermediate and the intermediate is at the place.

```
[show-at <=
     (>=> (show (at =thing =place))
          (show (at thing =intermediate)
               (#then (show (at intermediate place)))))]
```

The actor show-at is simply transitivity of at.

M. LOGIC AND PLANNING

"It is behavior, not meaning that counts."

Denotational semantics as formalized by Tarski for the quantificational calculus is one of the crowning achievements of mathematical logic. It has clarified the semantics of ordinary mathematical theorems and led to the development of model theory which is a flourishing mathematical field in its own right. We contend that it is less satisfactory as a semantic base for a theory of action and change. In this paper we formulate the beginnings of a semantic theory based on behavior instead of denotation. We then make some preliminary remarks on the relationship between behavioral semantics and denotational semantics.

A satisfactory theory for the representation of knowledge should have one unified totally integrated formalism and semantics. For example we should not have one formalism and semantics for expressing declaratives and a separate formalism and semantics for expressing procedures. For some years now we have been working to achieve this goal. The record of our progress is published in the Proceedings of the International Joint Conferences on Artificial Intelligence beginning with the first conference in 1969. In the course of this research we have developed the Thesis of Procedural Embedding of Knowledge which is that "Knowledge of a domain is intrinsically bound up with the procedures for its use." An important corollary is that the fundamental technique of artificial intelligence is Automatic Programming and Procedural Knowledge Base Construction.

We would like to show how the behavior of formulas in the quantificational calculus using actors and how the rules of natural deduction follow as special cases from the mechanism of extension worlds. In this way we can demonstrate how DEDUCTION is a special case of COMPUTATION.

"Is Model theoretic TRUTH a sufficient foundation on which to Base semantics for the Representation of Knowledge?"

The model theoretic definition of TRUTH for the quantificational calculus formalized by Tarski [denotational semantics] is very smooth but we contend that it glosses over semantic distinctions that are crucial for the representation of knowledge.

We find that the deductions of plans in PLANNER often carry more conviction [in the sense of Richard Weyhrauch] than proofs in the quantificational calculus. This is because our minds are better at grasping the constructive relationships

between the plans than the global noneffective relationship established by asserting that a set of axioms is true. Two plans can affect each other only if there is a causal chain of "wheels and cogs" connecting each other. These causal chains are formalized in the definition of HISTORY for actors given above. We seem to be able to design, control, and debug sets of plans better than sets of axioms. The history of Russell's Paradox and the question of the independence of the Axiom of Choice illustrate some of the kinds of problems with denotational semantics. Our point is further illustrated by the several inconsistent formulations of the "Blind Hand Problem" that have been produced in the quantificational calculus. Their inconsistency has been discovered almost by accident as proofs by contradiction get shorter and shorter until the negation of the consequence is found to be superfluous to the proof! People are quite tolerant of minor inconsistencies and the inability to tolerate any inconsistency at all in formulating problems is a sign of excessive semantic rigidity. In general we feel that a contradiction is evidence for a bug in one's plans or in the plan which is being constructed and that to satisfactorily resolve the bug it may be necessary to examine all the assumptions being made instead of only the most recent one. Currently there are no good ways to debug sets of axioms whereas there is a well established and rapidly developing technology for debugging procedures.

Another symptom of the problems with denotational semantics has been its failure to capture the notion of intuitive semantic entailment. Given that the moon is not made of green cheese, the following proposition is valid in the quantificational calculus:

"'The moon is made of green cheese' implies 1+1≠2"

Worse yet, the following sentence is also valid:

"'The moon is not made of green cheese' implies 1+1=2"

The problem is that denotational semantics defines (X IMPLIES Y) solely in terms of the denotation [truth value] of X and Y. Instead of insisting on a causal connection from X to Y. Logical implication is a useful concept in its own right and we will formulate its behavior below; but it is a serious limitation if stronger more intuitive forms of entailment cannot be semantically defined. The most natural way to write these semantic entailments appears to be as Procedural plan schemata that implement particular causal entailments.

We contend that deduction is best regarded as a special case of computation. Consider a formula of the form (every phi) which means that for every x we have that (phi x) is the case. The procedural meaning of the formula is a PLANNER SCHEMA for how it can be used. The formula has two important uses: it can be asserted and it can be proved.

Our behavioral definitions are reminiscent of classical natural deduction except that we have four introduction and elimination rules [PROVE, DISPROVE, ASSERT, and DENY] to give us more flexibility in dealing with negation.

"Then Logic would take you by the throat,
and force you to do it!"
Lewis Carroll

```
[every <=
    (=> [=phi]
      (cases
            [(=> (#prove)
                 (let
                     {[g = (anonymous)]}
                     (assert (object =g)
                         (#then (prove (phi g))))))
            (=> (#disprove)
                (disprove (phi =x)))
            (=> (#assert)
                (assert
                    (>=> (assert (object x))
                         (assert (phi x)))))
            (=> (#deny)
                (let
                   {[g = (anonymous)]}
                    (assert (object g)
                        (#then (deny (phi g))))))
            (=> (#display =s)
                (s
                    (print-open "(")
                    (print-string "every")
                    (print phi)
                    (print-close ")")))]))]

[some <=
    (=> [=phi]
        (cases
            [(=> (#prove)
                 (prove (phi =x)
                     (#then (prove (object x)))))
             (=> (#disprove)
                 (let
                     {[g = (anonymous)]}
                      (assert (object g)
                         (#then (disprove (phi g))))))
             (=> (#assert)
                 (let
                     {[g = (anonymous)]}
                      (assert (object g)
                          (#then (assert (phi g))))))
              (=> (#deny)
                  (assert
                      (>=> (assert (object =x))
                           (deny (phi x)))))
              (=> (#display =s)
                  (s
                      (print-open "(")
                      (print-string "some")
                      (print phi)
                      (print-close ")")))]))]
```

```
[and <=
    (=> (#and =conjuncts)
        (cases
           ((=> (#prove)
                (rules conjuncts
                    [(=> (empty)
                         (done))
                     (=> (#and =conjunct =rest-conjuncts)
                         (all-conjuncts
                           (#conjuncts
                             (prove conjunct)
                             (prove (%and rest-
                                    conjuncts%)))))])))
            (=> (#assert)
                (rules conjuncts
                    [(=> (empty)
                         (done))
                     (=> (#and =conjunct =rest-conjuncts)
                         (all-conjuncts
                           (#conjuncts
                             (assert conjunct)
                             (assert (%and rest-
                                    conjuncts%)))))]))
            (=> (#disprove)
                (rules conjuncts
                    [(=> (empty)
                         (not-disproveable))
                     (=> (#and =conjunct =rest-conjuncts)
                         (some-disjunct
                             (#disjuncts
                                 (extend-world (disprove
                                                 conjunct))
                                 (disprove
                                     (%and rest-
                                     conjuncts%)))))]))
            (=> (#deny)
                (rules conjuncts
                    [(=> (empty)
                         (not-deniable))
                     (=> (#and =conjunct =rest-conjuncts)
                         (some-disjunct
                             (#disjuncts
                                 (extend-world (deny conjunct))
                                 (deny
                                     (%and rest-
                                         conjuncts%)))))]))
            (=> (#display =s)
                (s
                    (print-open "(")
                    (print-string "and")
                    (print-element conjuncts)
                    (print-close ")")))))))]
```

```
[or <=
    (=> (#or =disjuncts)
        (cases
           {(=> (#prove)
                (rules disjuncts
                    [(=> (empty)
                        (not-proveable))
                    (=> (#or =disjunct =rest-disjuncts)
                        (cover-splits
                            (#disjuncts
                                (prove disjunct)
                                (prove (%or rest-
                                        disjuncts%)))))]))
            (=> (#assert)
                (rules disjuncts
                    [(=> (empty)
                        (not-assertable))
                    (=> (#or =disjunct =rest-disjuncts)
                        (make-splits
                            (#disjuncts
                                (extend-world (assert
                                            disjunct))
                                (assert (%or rest-
                                        disjuncts%)))))]))
            (=> (#disprove)
                (rules disjuncts
                    [(=> (empty)
                        (done))
                    (=> (#or =disjunct =rest-disjuncts)
                        (all-conjuncts
                            (#conjuncts
                                (disprove disjunct)
                                (disprove (%or rest-
                                        disjuncts%)))))]))
            (=> (#deny)
                (rules disjuncts
                    [(=> (empty)
                        (done))
                    (=> (#or =disjunct =rest-disjuncts)
                        (all-conjuncts
                            (#conjuncts
                                (deny disjunct)
                                (deny (%or rest-
                                        disjuncts%)))))]))
            (=> (#display =s)
                (s
                    (print-open "(")
                    (print-string "or")
                    (print-element disjuncts)
                    (print-close ")"))))))]
```

```
[intuitionist-not <=
   (=> [=phi]
       (cases
          {(=> (#prove)
               (disprove phi))
           (=> (#assert)
               (deny phi))
            (=> (#display =s)
               (s
                   (print-open "(")
                   (print-string "intuitionist-not")
                   (print phi)
                   (print-close ")")))}))]
```

We find ourselves convinced that the plans defined above have generally useful behaviors and would expect them to be a standard part of any actor system. However the plan defined below does not carry the same conviction that it always proceeeds in a useful or justifiable manner:

```
[classical-not <=
   (=> [=phi]
       (cases
          {(=> (#prove)
               (disprove phi))
           (=> (#disprove)
               (prove phi))
           (=> (#assert)
               (deny phi))
           (=> (#deny)
               (assert phi))
            (=> (#display =s)
               (s
                   (print-open "(")
                   (print-string "classical-not")
                   (print phi)
                   (print-close ")")))}))]
```

We find that it is extremely dubious that it is always permissible to DENY (NOT phi) simply by ASSERTING phi. To be able to DISPROVE (NOT phi) by simply being able to PROVE phi is equally unconvincing.

"Garbage in--garbage out."

Even with the dubious principle embodied in our definition of CLASSICAL-NOT we still haven't defined all the behavior that the quantificational calculus considers valid. In the quantificational calculus from (and theta (not theta)) every statement is deducible no matter how nonsensical! The following plan schemata realize this behavior [although in our view this behaviour is usually quite harmful]:

```
[gigo <=
    (>=> (prove =phi)
        (prove
           =theta
```

```
            (#then (prove (not theta)))))]
```

In certain contexts we would be willing to accept the following plan schemata although it does not always preserve causal chains.

```
[indirect-proof <=
    (>=> (prove (not =phi))
        (extend-world
            (assert phi
                (#then
                    (prove =theta
                        (#then (prove (not theta))))))))]
```

In a similar vein we feel that there are two methods for defining sets that carry conviction:

By specifying a <u>generating</u> procedure which can generate all the elements of the set.

By specifying a <u>deciding</u> procedure which is capable of deciding for any given actor whether it is a member of the set or not.

In some theories of computation [e.g. recursive function theory], it can be shown that the second is a special case of the first. However, we do not see any reason to suppose that it is <u>always</u> possible to generate <u>all</u> actors.

We would like to show how to use the above definitions to prove a simple theorem of the quantificational calculus: "If for some x such that for every y we have (p x y) then for every y there is some x such that (p x y)."

```
(implies
    (some
        (=> =x
            (every
                (=> =y
                    (p x y)))))
    (every
        (=> =y
            (some
                (=> =x
                    (p x y))))))
```

The proof is accomplished in the following way:

Create an extension of the current world and call it reality

```
    assert (some (=> =x (every (=> =y (p x y))))) in reality
    let x = anon1
        assert (object anon1)
        assert (every (=> =y (p anon1 y))) in reality
        assert (>=> (assert (object =y))
                    (assert (p anon1 y)))
        prove (every (=> =y (some (=> =x (p x y)))))
```

```
put (prove (every (=> =y (some (=> =x (p x y))))))
in utopia
let y = anon2
    assert (object anon2)
    assert (p anon1 anon2)
    prove (some (=> =x (p x anon2)))
    put (prove (some (=> =x (p x anon2)))) in utopia
    prove (p ? anon2)
```

We can distinguish several different uses for extension worlds:

1. World Directed Invocation

The extension world machinery provides a very powerful invocation and parameter passing mechanism for procedures. The idea is that to invoke a procedure, first grow an extension world; then do a world directed invocation on the extension world. This mechanism generalizes the previous pattern directed invocation of PLANNER-67 several ways. Pattern directed invocation is a special case in which there is just one assertion in the wish world. World Directed Invocation represents a formalization of the useful problem solving technique known as "wishful thinking" which is invocation on the basis of a fragment of a micro-world. Terry Winograd uses a special case of world-directed invocation using restriction lists in his thesis version of the blocks world. Suppose that we want to find a bridge with a red top which is supported by its left-leg and its right-leg both of which are of the same color. In order to accomplish this we can call upon a genie with our wish as its message. The genie uses whatever domain dependent knowledge it has to try to realize the wish.

```
(realize
    (utopia
        (#specs
            (color =top red)
            (supported-by =top =left-leg)
            (supported-by =top =right-leg)
            (left-of =left-leg =right-leg)
            (color =right-leg =color-of-legs)
            (color =left-leg =color-of-legs))))
```

2. Logical Hypotheticals

For example to prove that (implies p q) we could define the following:

```
[implies <=
    (=> [=the-antecedent =the-consequent]
        (cases
           {(=> (#antecedent)
                the-antecedent)
            (=> (#consequent)
                the-consequent)
            (=> (#prove)
                    ;"to prove something of the form (implies
```

```
            antecedent consequent)"
        (extend-world
            (assert
                the-antecedent
                (#then (prove the-consequent)))))
    (=> (#detach)
            ;"to detach a formula from (implies the-
            antecedent the-consequent) it must match
            the antecedent"
        (=> the-antecedent
                the-consequent))
    (=> (#assert)
        (assert
            (>=> (assert the-antecedent)
                (assert the-consequent))))
    (=> (#disprove)
        (extend-world
            (assert the-antecedent
                (#then (disprove the-consequent)))))
    ;there is no (#deny) clause because Ben Kuipers
    ;found a bug in the one we proposed and we
    ;couldn't find a substitute that carried conviction
                        }))]
```

By the Normalization Theorem for intuitionistic logic the above definition of implies is sufficient to mechanize logical implication. The rules of natural deduction are a special case of our rules for extension worlds and our procedural definition of the logical connectives.

3. Alternative Worlds

```
(let
    {[hell = (after-world-war-III world-1973)]}
      (compare-and-contrast world-1973 hell))
```

4. Perceptual Viewpoints

Perceptual Viewpoints can be mechanized as extension worlds. For example suppose rattle-trap is the name of a world which describes my car. Then (front rattle-trap) could be a world which describes my car from the front and (left rattle-trap) can be the description from the left side. We can also consider a future historian's view of the present by (view-from-1984 world-of-1972). Minsky [1973] considers these possibilites from a somewhat different point of view.

N. GENERAL PRINCIPLES

The following general principles hold for the use of extension worlds:

Each independent fact should be a separate assertion.

For example to record that "the banana banl is under the table tabl" we would assert:

```
(banana banl)
(table tabl)
(under banl tabl)
```

instead of conglomerating [McDermott 1973] them into one assertion:

```
(at
    (the banl (is banl banana))
    (place
        (the tabl (is tabl table))
        under))
```

A person knowing a statement can be analyzed into the person believing the statement and the statement being true. So we might make the following definition of knowing:

```
[know <=
    (=> [=person =statement]
        (and
            (believes person statement)
            (true statement)))]
```

Thus the statement [Moore 1973] "John knows Bill's phone number" can be represented by the assertion:

```
(knows John (phone-number Bill pn0005))
```

where pn0005 is a new name and (phone-number Bill pn0005) is intended to mean that the phone number of Bill is pn0005. The assertion can be expanded as follows:

```
(believes John (phone-number Bill pn0005))
(true (phone-number Bill pn0005))
```

However, the expansion is optional since the two assertions are <u>not</u> independent of the original assertion.

> "Whatever <u>Logic</u> is good enough to tell me is worth <u>writing down</u>," said the Tortoise. "So enter it in your book, please."
> Lewis Carroll

Each assertion should have justifications [derivations] which are also assertions and which therefore ...

Extraneous factors such as time and causality should <u>not</u> be conglomerated [McDermott 1973] into the extension world mechanism. Facts about time and causality should also be separate assertions. In this way we can deal more naturally and uniformly with questions involving more than one time. For example we can answer the question "How many times were there at most two cannibals in the boat while the missionaries and cannibals were crossing the river?" Also we can check the consistency of two different narratives of overlapping events such as might be generated by two people who attended the same party. Retrieval from data bases actors takes facts about time and causality into account in the retrieval. Thus we still effectively avoid most of the frame problem of McCarthy. The ability to do this is enhanced by the way we define data bases as actors.

AUTOMATIC PROGRAMMING

PUBLICATIONS

1. Bishop, Peter and Carl Hewitt, "Planner Reference Manual for the MULTICS Implementation," Version 1, Planner Technical Report No. 2, September 28, 1972.

2. Dertouzos, M. L., M. Athans, R. N. Spann, S. Mason, Systems, Networks and Computation: Basic Concepts, (McGraw-Hill, 1972).

3. Dertouzos, M. L., "Time Bounds on Space Computations," IEEE Transactions on Computers, Vol. C-22, No. 1, January 1973, pp. 12-17.

4. Fateman, Richard J., "Solution to Problem Number 2," (MACSYMA), ACM SIGSAM Bulletin, Bulletin No. 24, October 1972, pp. 12-13.

5. Fateman, Richard J., "Rationally Simplifying Non-rational Expressions," ACM SIGSAM Bulletin, Bulletin No. 23, July 1972, pp. 8-9.

6. Fateman, Richard J., Reply to an Editorial (concerning LISP), ACM SIGSAM Bulletin, Bulletin No. 25, March 1973.

7. Ginzberg, M. J., "Status of the Simulator in Protosystem I," Automatic Programming Group Internal Memo No. 3, July 1972.

8. Ginzberg, M. J., "Translation of Detailed System Simulation Language (DSSL) to Data Set Language (DSL) in Protosystem I," Automatic Programming Group Internal Memo No. 5, September 12, 1972.

9. Hewitt, Carl, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, AI TR 258.

10. Hewitt, Carl, Models of Procedure and the Teaching of Procedures, in: Some Current Views on Language.

11. Hewitt, Carl, et al., A Universal Modular ACTOR Formalism for Artificial Intelligence, PLANNER Technical Report No. 3, December 1972 (Revised March 1973 and June 1973).

12. Hewitt, Carl, et al., Actor Induction and Meta-Evaluation, Journal of the ACM-SIGPLAN Symposium on Principles of Programming Languages, Boston, Mass., October 1973.

13. Jessel, G. P., "A Theory of Computer-Aided Network Analysis," August 1972.

14. MACSYMA Primer - Introductory Section, Project MAC, M.I.T., August 1972.

15. MACSYMA Primer - Section 2: Trigonometric Functions, Project MAC, M.I.T., August 1972.

PUBLICATIONS continued

16. MACSYMA Reference Manual, Version Four, Project MAC, M.I.T., April 1973.

17. MACSYMA Reference Manual, Version Five, Project MAC, M.I.T., June 1973.

18. Mark, W., "Handling Goal-Structured Models," Automatic Programming Group Internal Memo No. 7, November 21, 1972.

19. Martin, W. A., Interactive Design in Protosystem I, Automatic Programming Group Internal Memo No. 4, August 21, 1972.

20. Martin, W. A. and Krumland, R., A Language for Describing Models of the World, Automatic Programming Group Internal Memo No. 6, October 17, 1972.

21. Martin, W. A., Krumland, R. and Sunguroff, A., More MAPL: Specifications and Basic Structures, Automatic Programming Group Internal Memo No. 8, February 7, 1973.

22. Martin, W. A., Translation of English into MAPL Using Winograd's Syntax, State Transition Networks, and a Semantic Case Grammar, Automatic Programming Group Internal Memo No. 11, April 17, 1973.

23. Morgenstern, M., "Automating the Design and Optimization of Information Processing Systems," Automatic Programming Group Internal Memo No. 10, February 16, 1973.

24. Moses, Joel, "Toward a General Theory of Special Functions," Communications of the ACM, Vol. 15, No. 7, July 1972, pp. 550-554.

25. Niamir, B., "Interactive Optimization of Information Processing Systems Represented in Data Set Language, Automatic Programming Group Internal Memo No. 9, February 1973.

26. Pless, Vera, "Power Moment Identities on Weight Distribution in Error Correcting Codes," (in) Blake, Ian (ed.), Algebraic Coding Theory, History and Development, (Dowden-Hutchinson-Ross, 1973).

27. Stinger, J. S., "Effective Computing Machines Using Inexact Substructures," July 1972.

OTHER RESEARCH

Academic Staff

Prof. V. Briabrin (visiting)
Prof. E. Fredkin
Prof. J. J. McCarthy (visiting)
Prof. M. Rabin (visiting)

DSR Staff

G. A. Briabrin
P. M. Gunkel
M. I. Levin
M. Pivar

Graduate Student

F. Manning
R. E. Sacks

Undergraduate Students

D. J. Morgan
M. J. Douglas

Guest

A. Endo

OTHER RESEARCH

During the period from October, 1972 till April, 1973 Victor Briabrin, a Visting Professor from Moscow, USSR was working at the Project MAC. His research was performed on the basis of the scientific exchange program between the National Academy of Sciences of the U.S.A. and the Academy of Sciences of the USSR. It included studying PLANNER, CONNIVER and related programming systems, with the purpose to make conclusions about the important features of the high-level programming languages used for the Artificial Intelligence and other advanced research in the computer science.

Part of his job was in establishing close contacts with the group involved in PLANNER implementation, in view of the parallel design of the similar programming system on the soviet computer in Moscow.

Intensive use of LISP on POP-10 was done in order to compare this programming system with the appropriate LISP implementation on the BESM-6 computer in Moscow. A possibility of transfering POP-LISP programs onto the BESM-6 was considered and the necessary adjustment of the BESM-LISP system has been outlined.

Besides studying CONNIVER, PLANNER and LISP, V.Briabrin participated in the series of Automatic Programming Group seminars conducted by Prof. W.Martin. The purpose was to study different programming techniques used for the general design and specific applications of the Automatic Programming Systems. Under the influence of the ideas which were discussed in the Automatic Programming seminars, V.Briabrin developed a model of an abstract research institute and described it in his paper [1]. An attempt at simulating a simple sociological structure has shown what are the basic relation types essential for creating a model and what are the best ways of knowledge representation in this specific domain.

Some aspects of model implementation were also considered, including construction of the general frame, filling it with the specific information and applying the request statements.

Another paper [2], written in Russian, was prepared by V.Briabrin for publishing in the Soviet Union. This paper contains the general survey of the Artificial Intelligence methods and their mixture with the systems programming technology in the field of creating advanced Automatic Programming Systems.

References.

1.An Abstract Model of Research Institute: Simple Automatic Programming Approach, Project MAC Memo, June 1973.
2.Artificial Intelligence and Automatic Programming (Russian) Computing Center, Academy of Sciences of the U.S.S.R. (to be published).

PROJECT MAC PUBLICATIONS

TECHNICAL REPORTS

* TR-1	Bobrow, Daniel G. Natural Language Input for a Computer Problem Solving System, Ph.D. Thesis, Math. Dept. September 1964	AD 604-730
* TR-2	Raphael, Bertram SIR: A Computer Program for Semantic Information Retrieval, Ph.D. Thesis, Math. Dept. June 1964	AD 608-499
TR-3	Corbató, 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
TR-14	Roos, Daniel Use of CTSS in a Teaching Environment November 1964	AD 661-807

PUBLICATIONS

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 L.
An Analysis of Time-Shared Computer Systems, Ph.D. Thesis, EE Dept.
June 1965
AD 470-715

TR-19 Russo, Francis J.
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 E.
CALCULAID: An On-Line System for Algebraic Computation and Analysis, S.M. Thesis, Sloan School
September 1965
AD 474-019

* TR-21 Denning, Peter J.
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 D.
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 B.
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 J.
OCAS - On-Line Cryptanalytic Aid System, S.M. Thesis, EE Dept.
May 1966 — AD 633-678

TR-28 Smith, Arthur A.
Input/Output in Time-Shared, Segmented, Multiprocessor Systems, S.M. Thesis, EE Dept.
June 1966 — AD 637-215

TR-29 Ivie, Evan L.
Search Procedures Based on Measures or Relatedness between Documents, Ph.D. Thesis, EE Dept.
June 1966 — AD 636-275

TR-30 Saltzer, Jerome H.
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. Thesis, Math. Dept.
September 1966 — AD 638-446

* TR-33 Norton, Lewis M.
ADEPT - A Heuristic Program for Proving Theorems of Group Theory, Ph.D. Thesis, Math. Dept.
October 1966 — AD 645-660

TR-34 Van Horn, Earl C., Jr.
Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. Thesis, EE Dept.
November 1966 — AD 650-407

* TR-35 Fenichel, Robert R.
An On-Line System for Algebraic Manipulation, Ph.D. Thesis, Appl. Math. (Harvard)
December 1966 — AD 657-282

* TR-36 Martin, William A.
Symbolic Mathematical Laboratory, Ph.D. Thesis, EE Dept.
January 1967 — AD 657-283

PUBLICATIONS

* 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. Thesis, EE Dept. August 1967	AD 662-224
TR-44	Gorry, G. Anthony A System for Computer-Aided Diagnosis, Ph.D. Thesis 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 663-504
TR-46	Alsop, Joseph W. A Canonic Translator, S.B. Thesis, EE Dept. November 1967	AD 663-502
* TR-47	Moses, Joel Symbolic Integration, Ph.D. Thesis, Math. Dept. December 1967	AD 662-666

TR-48 Jones, Malcolm M.
Incremental Simulation on a Time-Shared Computer, Ph.D. Thesis, Sloan School
January 1968 AD 662-225

TR-49 Luconi, Fred L.
Asynchronous Computational Structures, Ph.D. Thesis, EE Dept.
February 1968 AD 677-602

* TR-50 Denning, Peter J.
Resource Allocation in Multiprocess Computer Systems, Ph.D. Thesis, 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. Thesis, 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, D. E., R. H. Stotz, D. T. Ross and J. E. Ward (ESL-R-356)
An Integrated Hardware-Software System for Computer Graphics in Time-Sharing
December 1968 AD 685-202

* TR-57 Morris, James H., Jr.
Lambda-Calculus Models of Programming Languages, Ph.D. Thesis, Sloan School
December 1968 AD 683-394

PUBLICATIONS

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. Thesis, 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. Thesis, EE Dept.
April 1969 — AD 689-305

TR-61 Baecker, Ronald M.
Interactive Computer-Mediated Animation, Ph.D. Thesis, EE Dept.
June 1969 — AD 690-887

TR-62 Tillman, Coyt C., Jr. (ESL-R-395)
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. (ESL-R-398)
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. Thesis, EE Dept.
October 1969 — AD 699-501

* TR-66 Beyer, Wendell T.
Recognition of Topological Invariants by Iterative Arrays, Ph.D. Thesis, Math. Dept.
October 1969 — AD 699-502

* TR-67 Vanderbilt, Dean H.
Controlled Information Sharing in a Computer Utility, Ph.D. Thesis, 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. Thesis, Sloan School
June 1970 AD 710-011

* TR-69 Gertz, Jeffrey L.
Hierarchical Associative Memories for Parallel Computation, Ph.D. Thesis, 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. Thesis, Math. Dept.
August 1970 AD 712-069

TR-74 Edelberg, Murray
Integral Convex Polyhedra and an Approach to Integralization, Ph.D. Thesis, 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. Thesis, EE Dept.
September 1970 AD 713-988

PUBLICATIONS

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. Thesis, 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. Thesis, ME Dept.
January 1971 — AD 717-951

* TR-82 Krakauer, Lawrence J.
Computer Analysis of Visual Properties of Curved Objects, Ph.D. Thesis, EE Dept.
May 1971 — AD 723-647

TR-83 Lewin, Donald E.
In-Process Manufacturing Quality Control, Ph.D. Thesis, Sloan School
January 1971 — AD 720-098

* TR-84 Winograd, Terry
Procedures as a Representation for Data in a Computer Program for Understanding Natural Languages, Ph.D. Thesis, Math. Dept.
February 1971 — AD 721-399

TR-85 Miller, Perry L.
Automatic Creation of a Code Generator from a Machine Description, Elec. E. Degree, EE Dept.
May 1971 — AD 724-730

TR-86 Schell, Roger R.
Dynamic Reconfiguration in a Modular Computer System, Ph.D. Thesis, EE Dept.
June 1971 — AD 725-859

TR-87 Thomas, Robert H.
A Model for Process Representation and Synthesis, Ph.D. Thesis, EE Dept.
June 1971 AD 726-049

TR-88 Welch, Terry A.
Bounds on Information Retrieval Efficiency in Static File Structures, Ph.D. Thesis, 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 Structures, Ph.D. Thesis, Math. Dept.
September 1971 AD 731-690

TR-92 Wang, Paul S.
Evaluation of Definite Integrals by Symbolic Manipulation, Ph.D. Thesis, Math. Dept.
October 1971 AD 732-005

TR-93 Greif, Irene G.
Indiction in Proofs about Programs, S.M. Thesis, EE Dept.
February 1972 AD 737-701

TR-94 Hack, Michel H. T.
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. Thesis)
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. Thesis, EE Dept.
March 1972 AD 739-678

PUBLICATIONS

TR-98 Stockmeyer, Larry J.
Bounds on Polynomial Evaluation Algorithms, S.M. Thesis, EE Dept.
April 1972 AD 740-328

TR-99 Lynch, Nancy A.
Relativization of the Theory of Computational Complexity, Ph.D. Thesis, 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 w-Automata, S.M. Thesis, EE Dept.
September 1972 AD 749-367

TR-103 Sekino, Akira
Performance Evaluation of Multi-programmed Time-Shared Computer Systems, Ph.D. Thesis, EE Dept.
September 1972 AD 749-949

TR-104 Schroeder, Michael D.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. Thesis, 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. Thesis, EE Dept.
April 1973 AD 760-001

TR-109 Johnson, David S.
Near-Optimal Bin Packing Algorithms
Ph.D. Thesis, Math. Dept.
June 1973 PB 222-090

TR-110 Moll, Robert
Complexity Classes of Recursive Functions
Ph.D. Thesis, Math. Dept.
June 1973

TR-111 Linderman, John P.
Productivity in Parallel Computation Schemata
Ph.D. Thesis, EE Dept.
June 1973

TECHNICAL MEMORANDA

TM-10 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 (Formerly Programming Linguistics Group Memo No.6, June 12, 1970)
September 1970 AD 712-068

* TM-13 Graham, Robert M.
Use of High Level Languages for Systems Programming (Formerly Programming Linguistics Group Memo No.2, November 20, 1969)
September 1970 AD 711-965

* TM-14 Vogt, Carla M.
Suspension of Processes in a Multi-processing Computer System (Based on S.M. Thesis, EE Dept., February 1970)
September 1970 AD 713-989

TM-15 Zilles, Stephen N.
An Expansion of the Data Structuring Capabilities of PAL (Based on S.M. Thesis, EE Dept., June 1970)
October 1970 AD 720-761

TM-16 Bruere-Dawson, Gerard
Pseudo-Random Sequences (Based on S.M. Thesis, EE Dept., June 1970)
October 1970 AD 713-852

TM-17 Goodman, Leonard I.
Complexity Measures for Programming Languages, (Based on S.M. Thesis, EE Dept., September 1971)
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

TECHNICAL MEMORANDA

TM-10 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 (Formerly Programming Linguistics Group Memo No.6, June 12, 1970)
September 1970 — AD 712-068

* TM-13 Graham, Robert M.
Use of High Level Languages for Systems Programming (Formerly Programming Linguistics Group Memo No.2, November 20, 1969)
September 1970 — AD 711-965

* TM-14 Vogt, Carla M.
Suspension of Processes in a Multi-processing Computer System (Based on S.M. Thesis, EE Dept., February 1970)
September 1970 — AD 713-989

TM-15 Zilles, Stephen N.
An Expansion of the Data Structuring Capabilities of PAL (Based on S.M. Thesis, EE Dept., June 1970)
October 1970 — AD 720-761

TM-16 Bruere-Dawson, Gerard
Pseudo-Random Sequences (Based on S.M. Thesis, EE Dept., June 1970)
October 1970 — AD 713-852

TM-17 Goodman, Leonard I.
Complexity Measures for Programming Languages, (Based on S.M. Thesis, EE Dept., September 1971)
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

PUBLICATIONS

* TM-20	Jones, Thomas L. A Computer Model of Simple Forms of Learning, (Based on Ph.D. Thesis, EE Dept., September 1970) 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 (Based on S.B. Thesis, ME Dept., May 1972) 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's 1-9 were never issued

*	Project MAC Progress Report I to July 1964	AD 465-088
	Project MAC Progress Report II July 1964-July 1965	AD 629-494
*	Project MAC Progress Report III July 1965-July 1966	AD 648-346
	Project MAC Progress Report IV July 1966-July 1967	AD 681-342
	Project MAC Progress Report V July 1967-July 1968	AD 687-770
	Project MAC Progress Report VI July 1968-July 1969	AD 705-434
	Project MAC Progress Report VII July 1969-July 1970	AD 732-767
	Project MAC Progress Report VIII July 1970-July 1971	AD 735-148
*	Project MAC Progress Report IX July 1971-July 1972	AD 756-689

****

Copies of all MAC reports listed in Publications may be secured for the National Technical Information Service, Operations Division, Springfield, Virginia, 22151. Prices vary. The AD number must be supplied with the request.

*Out of print, may be obtained from NTIS (see above).