1976–77

Computer Science and Computer Engineering Research Review

a report published by the Departments of Computer Science and Electrical Engineering
University of Rochester
Rochester, New York
# TABLE OF CONTENTS

3 Introduction
   Robert L. Sproull

Selected Research Articles

5 . . . . RIG, Rochester’s Intelligent Gateway: System Overview
   Eugene Ball and Richard Rashid

12 . . . . The CERF Computer System
   Neil Wilhelm, David Pessel, and Charles Merriam

17 . . . . Geometric Modelling of Mechanical Parts and Processes
   Herbert Voelcker and Aristides Requicha

29 . . . . Automatic Data Structure Selection
   Jerome Feldman, James Low, and Paul Rovner

34 Seminar Meetings

37 Publications

39 Grants
INTRODUCTION

Many readers will have managed their lives very well up to this point without having given much thought to the University of Rochester. A few words about the University, the setting for the ambitious young Computer Science and Computer Engineering programs described in this document, may therefore be helpful.

We are a small, private university. For the first seventy years, we were really only one of the many excellent liberal arts colleges in the Northeast, although we called our institution a university from the beginning. In the early Twenties, the addition of two colleges initiated the transition to a major university: the Eastman School of Music, perhaps the most renowned conservatory, imbedded in a university, and the School of Medicine and Dentistry, one of the first “Flexner” (science-based) medical schools. During the Thirties and (especially) the Forties and Fifties, departments in the liberal arts college developed into full Ph.D.-granting university departments, and that college spawned three additional professional schools.

We are small and yet we intend to be among the very best in everything we undertake. A corollary is, therefore, that we are highly selective in what we undertake. We claim to be eclectic without being chaotic. Another corollary of modest size is the opportunity to articulate and integrate programs and disciplines naturally. Although cooperative, experimental, or interdisciplinary programs can also be accomplished even in the canonical state university, they often must be so delicately contrived that flexibility is lost. The Computer Science Department is by no means the only or even the most recent example of programs here that take advantage of the flexibility and naturalness of interdepartmental and intercollege associations. I could cite biomedical engineering, the Center for Visual Science, the Institute of Optics, the Rochester Plan (an integrated premedical, medical, health sciences program), the Laboratory for Laser Energetics, and many more.

The Computer Science Department, although young as departments go, is already woven into the fabric of the University. From the beginning it was conceived as a permanent department, one of the central strengths of the University. The Computer Engineering program (of the Electrical Engineering Department) has become a major component of the College of Engineering and Applied Science. A key reason that these programs have become centrally important is the messianic view adopted by the initial staffs. They believe, in common with almost everyone, that computers as tools have had and will have a major impact on almost all aspects of life. The messianic view goes beyond that common view in holding that computer science provides a set of intellectual tools which will have equally profound effects on education, research, and scholarship. The most important of these intellectual tools are ways of talking and thinking about processes. The writing and perfecting (“debugging”) of computer programs have forced computer scientists and engineers to learn to deal with complex, time-dependent processes in a most general and powerful way. The languages, intuitions, and mathematical techniques developed (and developing) in this effort seem to provide important new ways of describing and reasoning about an extremely broad range of topics. Many other groups within the University share this view and have formed collaborative arrangements with computer scientists and engineers. Even viewed at this early stage, it seems clear that the Rochester environment is especially fruitful for these articulated efforts.

This research review describes a few of the initial projects of the new computer programs and provides references to some others. I hope it gives you the “flavor” of our programs and attracts your interest.

Robert L. Sproull
President
RIG, ROCHESTER'S INTELLIGENT GATEWAY: SYSTEM OVERVIEW

Eugene Ball and Richard Rashid

"When I opened my eyes I saw the Aleph... the place where, without any possible confusion, all the places in the world are found, seen from every angle."
—Jorge Luis Borges, "The Aleph" (translation by Anthony Kerrigan)

In a world where networks of diverse computing resources are growing and intertwining, there is a need for systems which provide access to a variety of computers and serve as intelligent gateways to their use. The Computer Science Department of the University of Rochester has designed and is implementing an intelligent gateway system, RIG, which will provide powerful tools for effectively using a wide range of computing resources.

The term "gateway system" has been coined to describe a computer system designed primarily to connect users to other computers and computer networks. An "intelligent gateway" goes beyond the simple function of communication. It plays a central role in the handling of information, actively translating the intentions of users into communication forms which can be understood by other computers. Because a "user" may in fact be another machine, an intelligent gateway can serve as a communication link between the various computers and computer networks to which it is connected.

The RIG system development effort has been divided into three major parts, each of which is described in greater detail later in this overview:

(a) An operating system kernel, called Aleph, has been developed. Aleph's primary function is to oversee the transfer of information and control between processes running within the system. A description of Aleph is given in Part One of this article.

(b) Server Processes have been developed to handle requests for resources available to the system. The use of independent processes to handle resource allocation allows RIG to provide intelligent access to its facilities. A description of the way in which Server Processes manage video display terminals is given in Part Two.

(c) The writing of User Processes has begun. These are the intelligent agents which translate user desires into requests to Server Processes. Part Three describes the broad outlines being followed in the design and implementation of these processes.

The strong emphasis on the use of independent processes to handle different RIG functions is based on a desire to keep RIG modular and easy to understand. For this same reason, RIG is being written almost entirely in BCPL [Curry, 75], an ALGOL-like systems programming language.

The physical layout of RIG, with its connections to various remote computing services, is shown in Figure 1. The resources available are similar in kind to those found in other research environments. Three large machine connections are shown: 360, KL10, and CERF. The first is the University batch-processing computer which

![Figure 1: RIG Hardware Configuration](image_url)
also has some interactive text editing and programming capabilities (WYLBUR, APL). The second is a general purpose time-sharing system. CERF (Computer Engineering Research Facility) [Wilhelm, 75] is an experimental computer designed and being built by the University of Rochester Department of Electrical Engineering (see next article). All these machines are located several hundred meters away from the gateway processor.

The two network connections shown (ARPANET, ETHERNET) represent two distinctly different kinds of computer networks. The ARPANET is a large nationwide network of research computers. The ETHERNET represents a high-speed (3 megahertz) in-house network of four 64K, 16-bit word mini-computers.

The RIG central processor is a Data General Eclipse—a relatively powerful mini-computer. A second processing unit, labeled DCU-50 in the figure, manages asynchronous and synchronous communication. It communicates with the Eclipse via shared memory and a cross-interrupt facility. Current plans call for the addition of a second Eclipse processor to the system in the near future.

RIG also provides facilities for local file storage on disk and magnetic tape and for printed and plotted output. This capability is basic to the gateway concept. RIG users should be able to create and edit files locally, with all the advantages that local computing offers: a single familiar editor, fast reliable response, and better security and protection. They may then process their files on any one of several larger machines.

1. ALEPH—A GATEWAY OPERATING SYSTEM

The computing activity within RIG logically divides itself into three levels: kernel, foreground and background. The distinction between these computing levels is based on the scheduling and communication requirements of the various functions performed by RIG (See Figure 2).

<table>
<thead>
<tr>
<th></th>
<th>Communications with</th>
<th>Interrupt Discipline</th>
<th>Runs to Completion?</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Foreground only</td>
<td>Standard Multi-Programming (=10 sec)</td>
<td>Yes, except for hardware interrupts</td>
<td>Always Available</td>
</tr>
<tr>
<td>Foreground</td>
<td>All</td>
<td>Polling (=1 m sec)</td>
<td>To a polling (clean) point</td>
<td>Very rapid access by map switch</td>
</tr>
<tr>
<td>Background</td>
<td>Foreground only</td>
<td>Check-pointing (=10 sec)</td>
<td>Preemptable</td>
<td>Secondary Storage</td>
</tr>
</tbody>
</table>

Figure 2: Properties of RIG processes at the three levels

a) The Aleph Foreground

The foreground is the locus of all RIG activity. RIG will be required to provide three kinds of service to the external world: full or half duplex character transmission between terminals and any of the gateway accessible computers, file transfer operations between any two systems or peripherals, and process to process communication.

Figure 3: Typical Aleph process organization
tion among systems. We have chosen to dedicate an independent process within RIG to each external connection and to establish a uniform message-passing system for inter-process communication. Each process has responsibility for maintaining its external communication protocol and for performing any conversions necessary to enable it to present a standard interface to the rest of the system. Thus, knowledge of the idiosyncrasies of a particular connection will be required only within the process dedicated to it. A typical collection of foreground processes is given in Figure 3. As an example, the ARPANET process depicted in Figure 3 has various "ports" of communication to other processes in the foreground. These correspond to "connections" between remote systems and processes within RIG. The use of standardized message formats for character, file, and process communications allows the high degree of flexibility and modularity necessary to provide practical communications with widely varying computer systems. The Aleph message discipline is also intended to support the distribution of RIG functions over several machines.

Foreground processes communicate through a message queueing and distribution system similar to that described in [Walden, 72]. Messages can also be sent to and from background jobs (to be described later). All processes reside in main memory while they are in use.

b) The Aleph Kernel

The Aleph kernel provides the basic functions used by all processes in the system. These functions include:

(1) interrupt management;
(2) scheduling of foreground and background processes;
(3) inter-process communication; and
(4) memory management and allocation.

Interrupt Management

Aleph device drivers are controlled by a conventional priority interrupt mechanism. A "device driver" is a function executed whenever a particular interrupt occurs. Device drivers may send messages to processes running in either the background or foreground, but they cannot receive messages. Each device driver usually has associated with it a single foreground process which handles requests for the use of that device.

Scheduling of Foreground and Background Processes

Control of foreground and background processes is based on a modifiable priority scheme. The Aleph scheduler selects the highest priority ready foreground process for execution. If all foreground processes are suspended, control passes to a single background job until the foreground requires processing. The scheduler uses no time-slicing or pre-emption in the foreground. Therefore the foreground is rescheduled only when the currently executing process relinquishes control explicitly. I/O interrupts do not cause rescheduling so the effects of an I/O-complete are not felt in the foreground until the next rescheduling.

A foreground process allows rescheduling in one of three ways:

1. By sending a message to another process;
2. By requesting a message from its input queue;
3. By performing a "clean point" call.

Clearly the performance of Aleph will depend upon the frequency of these rescheduling requests. Because the system functions primarily as an input/output distribution network, very little processing occurs without the necessity of inter-process communication. This means that most processes time-slice themselves by continually sending or receiving messages. In situations where processing time between messages may be great, each process is required by convention to periodically relinquish control. Since the Aleph foreground level is not intended to include programs written by naive users (such programs will be executed on remote machines) these behavioral requirements can reasonably be made and met.

Simplicity is the greatest advantage to be gained by designing a request-driven scheduling system isolated from hardware interrupts. The overhead of context switching, for interrupts or rescheduling, is reduced to a minimum. More importantly, because each process knows that it loses control only at its own request, it can avoid the synchronization and critical race problems that would arise if it were being time-sliced.

Inter-Process Communication

Communication between processes in the foreground is in the form of "messages." Each process has a unique process number and any other process can send it a message using this number as an address. The routines that support the message system are included in the Aleph kernel. These routines maintain an input queue of messages for each process in the foreground. When a process requests a message Aleph removes one from its input queue and returns it to the caller. If there are no messages waiting, the caller is suspended until a message is sent to it by another process. Sending a message causes it to be linked into the receiver's queue, and awakens the receiver if it has been suspended by a message request.

The address used to specify the destination (or source) of a message consists of a process number and a port number. The port mechanism allows a process to define several logical addresses within itself and communicate with other processes from each of them (Figure 4). These ports are normally allocated for communication concerning a single set of requests. For example, a file system process might allocate a port to represent a single open disk file. All messages requesting manipulation of that file would be sent to that port of the file system process.

Since Aleph allows a process to receive the next message for a specified input port, processes can easily control their own input by assigning ports to logical tasks and
can allow the message switching system to perform any queuing that might be required.

Aleph also allows a process to wait for a message to arrive from a specified process-port. Thus a process can suspend itself until it has received an acknowledgement that a critical request has been completed, thereby allowing the processing of other messages to continue.

In addition to these message primitives, Aleph allows each process to inspect messages waiting in its input queue and to receive a particular message from any place in the queue. When requesting a message, a process also has the option of specifying a timeout period, and if no message arrives within that time, Aleph will notify it that a timeout occurred. This feature enables the system to recover from error situations in which a process fails to respond to requests.

The Aleph message protocol confines interactions between system modules to a well-structured format that discourages poorly defined dependencies between processes. Because of this structure, Aleph is able to provide two important ways for modifying information flow within the system. The “shadow” facility defines a shadow process which receives a copy of every message sent to a specified target process. The shadow process can then monitor its activity, gather statistics, maintain logs, etc., without requiring any modification of the target process. The shadow facility can also be used to debug new versions of system modules. The “interposer” facility allows a process to intercept all messages sent to a particular process or to intercept all messages originating from it. Thus, an interposer could intercept all transactions between a pair of processes and perform additional processing on the messages without requiring any changes in the code of either process. This ability to add “intelligent” processing to any information path in the system provides the flexibility necessary for a gateway system.

Figure 4: Aleph processes: Use of port numbers

Memory Management

The Aleph memory manager (included in the kernel) maintains memory pools from which space for data buffers, system tables, and stack is allocated. Memory in each pool is allocated using a boundary tag system. Each buffer has two “handles” that can be allocated to Aleph foreground processes, thus giving them access to the buffer. Only two processes can have simultaneous access to a buffer, but any process can elect to transfer its access rights to another process. A buffer is returned to the memory pool only after both of the processes having access to it release their handles. In a typical example, the tape drive process might ask the file system for the next block of an open file. The file system process would allocate a buffer with handles for itself and the tape process. After filling the buffer, it would send the tape process a message that the transfer was complete and release its own handle. When the tape process was finished with the data it would release its handle and the buffer would be re-allocated.

For still larger data transfers, two processes can allocate a group of buffers that pass back and forth between them. This scheme requires the overhead of buffer allocation only once, and simplifies the implementation of a multiply buffered transfer strategy. Because only two processes have access to any buffer at one time, synchronization problems involving buffers are relatively local and easy to understand.

c) The Aleph Background

Aleph should be able to maintain rapid response to a large number of widely varied requests. To insure this, we must explicitly relegate some longer operations to a background level. Typical background tasks include file transfer, long editor searches, etc., as well as traditionally batch operations such as compiling. Thus a program setting up a file transfer will be in the foreground, but the transfer itself will not have guaranteed response and can be done in the background. It is currently planned that background tasks will normally run to completion. There will be some pre-emptive checkpointing of background tasks, but this should be kept to a minimum.

2. FOREGROUND SYSTEM FUNCTIONS:
RIG SERVER PROCESSES

By the nature of RIG’s design, many of the functions commonly considered part of an operating system kernel are Server Processes running in the RIG foreground. Among these are file system maintenance and peripheral management. This improves RIG’s overall flexibility by increasing its modularity. As an example of the way in which Server Processes are used, this section describes the management of video display terminals in RIG. Other RIG Server Processes, such as the various file system handlers, command interpreters, and editors, will be discussed in separate reports.
**a) Screen Management**

It is important that a gateway allow its users to oversee several tasks in varying stages of completion. Many RIG tasks (e.g., file transfers, compilations, the monitoring of number crunching programs on host computers, etc.) require relatively little supervision and yet would take a disproportionate amount of working time away from the user if nothing else could be done while they were in progress.

While not commonplace, mechanisms which allow users to oversee several tasks have been included in other systems (e.g., Tenex Telnet). The problem encountered by previous implementations, however, has been one of informing the user about the progress of his activities. Typically the user has only one line of communication and may view the output of only one process at a time. This forces him to periodically look in on each process to assess its state. His working context is at best a long scroll of paper and at worst his fragile short-term memory. Consequently, although it might increase his useful terminal time, a user may not fully use such a facility simply to avoid being burdened by its complexity.

**b) Screens, Regions, and Subregions**

The advent of inexpensive display oriented terminals has suggested a solution to this problem: divide the user's visible screen into logical regions, each of which may be dedicated independently to different user tasks. Output could thereby be spatially as well as temporally related, permitting the user to view the activity of more than one program at a time [Swinehart, 74].

This solution has been adopted in RIG. A special process called the Screen Handler is responsible for the allocation of space on each terminal display screen.

A "screen" is defined to be those lines of text physically visible on a display device. A "region" is a rectangular area within a screen in which the output of a single task may be displayed. A "subregion" is a rectangular area within a region which is independent of all other areas of the screen. Subregions are the basic unit of screen space allocation. A region is simply a collection of contiguous subregions belonging to the same Aleph process, and a screen is a collection of regions visible on the same display device. No empty space is allowed in the sense that the physical screen is always allocated completely to one or more regions, each of which in turn consists of one or more subregions. New regions and subregions may be created only by splitting existing entities into two parts of variable size.

This hierarchical division of the screen permits the output requirements of several independent processes to be satisfied without compromising the readability of the display. An Executive process is associated with each user in the system. This process is the root of the tree of processes spawned by the user of the system and as such is the holder of special privileges in communication with the Screen Handler. The Executive alone, presumably at the request of the user, performs the task of sectioning the screen into regions. These regions are allocated to the processes spawned by the Executive which may then divide them into subregions as necessary. The result is positional constancy and spatial integrity of information. Regions of the screen devoted to user processes do not vary in size or change position except at the explicit request of the user acting through the Executive. Moreover, all output pertinent to the execution of a single process is contiguous on the screen. A typical "snapshot" of a screen would look like Figure 5.

![Figure 5: Sample use of video display by multiple RIG processes](image-url)
A serious drawback to the use of the screen to effectively inform the user about the state of his work is the relatively small amount of displayable text available on most terminals. Generally speaking, the entire context of a user's activities cannot always be visible on a single screen. Thus it becomes crucial for the mapping of lines to areas of the screen to be flexible. Important text, as selected by the user or his program, should always be visible, but less recent text should not necessarily be lost. It is for this reason that RIG isolates the function of terminal output from the task of collecting the outgoing information of an Aleph process. No process ever directly sends a message to the terminal output handlers. Instead, an Aleph process modifies a data structure called a "pad." If the pad is mapped into a screen subregion, this modification will also be made to the display.

c) Pads

A pad is both a data structure and a process. As a data structure it consists of a variable number of text lines which could be used, for example, as an input buffer, a window onto a file, or a depository for temporary textual data. As a process a pad has the ability to send and receive messages. Through messages it provides a set of text manipulation primitives to the foreground process which owns it, including most of the functions performed by simple text editors, e.g., character and line insertion and deletion, searching and substitution, scrolling, overwriting, and others. Protection is afforded to the system because all access to pad data structures is constrained. At the same time RIG processes are provided with a powerful tool for text manipulation.

Pads can also be used for communication between Aleph processes and a user terminal. The textual content of a pad can be mapped onto any subregion of a screen. When this is done, all changes made to the pad are automatically reflected in changes made to its terminal image. Because a pad may contain more text than can be displayed in a given subregion, each pad maintains an internal pseudo-cursor which points to the current focus of attention (either as indicated by a cursor motion message sent by the foreground process which owns the pad or by the last change made to the pad). Only that portion of text about the pad pseudo-cursor which will fit in the subregion is displayed. Using pad pseudo-cursor motion messages, the owning foreground process can select which portion of the pad is to be displayed, providing a working context through pad commands rather than screen space (a scarce resource!).

d) Pads and Files

In order to facilitate the use of a pad as a textual window onto a file, each pad can be provided with an input process-port by its owning process. Whenever pad underflow occurs, a message will be sent to this port requesting a new line of text. In the most common case, the input process-port will correspond to an open file being handled by an Aleph file process (Figure 6). The amount of a pad data structure which is kept in main memory is small and depends on system load; the rest is kept on special disk scratch files. This allows a pad to be used for such tasks as text editing or keeping transcripts of user sessions for later playback or printing. At any time the entire contents of a pad may be transferred to a permanent disk file.

3. INTELLIGENT GATEWAY FUNCTIONS

One way to view the intelligent gateway concept is that a user should be able to access a remote subsystem without knowing the operating system conventions of the machine on which the subsystem is running. There are four principal components to such an intelligent terminal system:

1. the user interface;
2. the data-structure (directory) for keeping track of system state;
3. a set of rules of procedure; and
4. a response handling capability.

We will discuss each of these briefly, describing its role and how we plan to implement it.

In most general terms, the user interface includes many pieces such as the editor, screen handler, etc. In the narrow sense considered here, the user interface is a command language. For an intelligent terminal system, one would like a more expressive and general language than is common in operating systems design. The user will want to provide descriptive information and rather complex instructions in addition to simple command sequences. We plan to use a compiler-like language and implement it using a production language (PL) interpreter [Gries, 71]. The PL system is natural to use, has a state-table nature, and can easily be made to accommodate different command languages.
The directory envisioned for RIG is an extension of the standard file directory which has a number of attributes such as location, size, format, protection, etc. A RIG directory extends this idea in two ways: (1) directory entries are much richer because of the need to deal with many different systems; and (2) entries are included for several entities (subsystems, parameters, keywords, etc.) in addition to files. From the point of view of a user, the directory structure is an associative store. We also expect to extend the notion of hierarchical directories to be more like the context mechanisms [Bobrow, 73] of the AI languages. The system should be able to be context sensitive in its handling of completion, checking, defaulting, etc.

The rules for handling any particular situation in the RIG environment are not too complex, assuming that one has a reasonable set of primitives. This gives rise to the hope that a user can be made to understand and perhaps change what is being done to him. Recent work in AI has yielded a number of ways of expressing sets of rules of procedure. These range from totally undirected axiom schemes, through situation-action (production) rules to very specific routines. Our model is of about the generality of Schank-Abelson scripts [Schank, 75]: there is an expected sequence of events and actions, but many details are expected to vary from case to case. Preliminary efforts suggest that we can build a readable yet efficient rule language for at least some simple intelligent terminal functions. The interpreter for this rule language will be implemented using the same PL system used for the command language.

Perhaps the most difficult task of an intelligent terminal system is responding appropriately to messages from other systems. One could, with some justice, treat this as a restricted natural language understanding task and use the pertinent methods. We are choosing to try a simpler scheme based (mirabile dictu) on a PL interpreter. It seems that if RIG can keep enough context, then it can use the appropriate set of PL tables and handle many responses without employing complex methods. We do not anticipate that casual users will learn to write response handlers, but it shouldn't really be very difficult in most cases.

Each of the four components has been designed to use the simplest adequate techniques. This makes the intelligent terminal effort seem more like one in systems programming than one in AI. Many of the really difficult problems will involve AI techniques (e.g., trying various methods for achieving a goal), but we feel that a great deal of understanding can be attained by constructing a system which is conceptually straightforward.

References
THE CERF COMPUTER SYSTEM
Neil Wilhelm, David Pessel, and Charles Merriam

In 1973 the Department of Electrical Engineering of the University of Rochester launched a computer engineering program to study the design and implementation of computer systems. Out of this effort grew the recognition of an important need—not met by computer manufacturers—for a flexible mechanism for investigating tradeoffs between computer hardware and software, particularly within operating systems. It is this need that motivated the design of our Computer Engineering Research Facility (CERF), a computer system that features high-speed multiprocessing and provides an experimental laboratory facility for operating systems research.

Most conventional computer operating systems are almost exclusively oriented towards the concept of single processor systems. Although some of these operating systems allow multiprogramming, little progress has been made with multiprocessing computer systems. Multiprocessing systems, however, offer the potential advantages of increased throughput and system availability despite failure of some of the processing units (Wulf & Bell, 1972; Baskin et al., 1972).

Two major questions, which present themselves in the design of multiprocessor computer systems, are:

1) How can operating systems be designed to efficiently handle processor scheduling, memory management, and file and I/O management on a multiprocessor system? An important aspect of this question concerns the management of the operating system itself: should it be executed by one dedicated processor or should its functions be distributed throughout the system in some fashion?

2) What are the appropriate tradeoffs between computer architecture and the operating system? Because of decreasing hardware costs, it is becoming increasingly possible to experiment with many ramifications of this issue, including the possibility of modifying hardware implementations once they are completed without a major cost to the system.

Our plan is to investigate these research areas with the assistance of our unique laboratory tool, the CERF computer system. This system and some of our research objectives are described on the following pages (and in greater detail in [Wilhelm et al., 1976]).

SYSTEM DESCRIPTION

The CERF computer system is depicted in Figure 1. The primary elements of the system are four Central Processing Units (CPUs). These are independent, microprogrammable processors, featuring 64-bit data paths and 72-bit microinstructions. Each processor has its own scratchpad of 64 (64-bit) registers; microinstructions are fetched from a common (and expandable) 1024-microinstruction storage. Each microinstruction features field extraction and branching capabilities, automatic stacking of subroutine return addresses to a depth of 16 levels, indirect register references with no time penalty, and a large selection of arithmetic and logical operators. The processor clock is 10 MHz, providing 400 ns microinstruction execution-time (all microinstructions take the same time).

The CPU's communicate to the external world via two data buses: the Stunt-Box Bus and the Memory Bus. The Stunt-Box Bus is used for the attachment of specialized hardware, such as multipliers, dividers, target machine instruction decoders, etc. The Memory Bus connects the CPU's to the main memory, which initially will be 128k bytes, arranged as 64-bit words, of solid state memory. Error detection and correction will be provided by an 8-bit Hamming code appended to each memory word.

I/O devices are divided into two categories: high speed and low speed. High-speed devices, such as a disc (an 80-megabyte moving-head disc) or a drum (a 1-megabyte fixed-head disc), are connected to the I/O Bus via two Selector Channels. These channels are extremely fast, special-purpose programmable computers with 16-bit words. Low-speed devices are connected through an Interdata 7/16 minicomputer with 16k bytes of 1-μs primary memory. The card reader, printer, and other miscellaneous low-speed peripherals are attached to the 7/16 by a special interface called the Multiplexor Channel. The CPU console is simulated by an interface to the Multiplexor Channel. Terminals and communication equipment are connected to the 7/16 by the Terminal Controller, which is a very fast programmable computer similar to the Selector Channels. The Terminal Controller also provides a path between the 7/16 and the primary memory of the CPU.

In addition to handling the low-speed peripherals, the 7/16 assists in debugging hardware via the simulated console, and, with its core memory, can be used as a convenient means of bootstrapping the system.

The desire to have multiprocessing capability with three or more processors poses a number of potential
Figure 1: Computer Engineering Research Facility (CERF) computer system

Figure 2: CERF computer system data flow
design problems such as processor priorities, interference, and lock-out. In addition, multiple processors often result in multiple copies of the same hardware, such as the register banks of CERF. These design problems can be solved conveniently by a technique that is similar to one used by Control Data Corporation [Thornton, 1964] for the peripheral processors in their CDC 6000 and 7000 series systems. The central processor hardware of CERF is divided into four disjoint subsets, corresponding to four stages of microinstruction execution. Thus, a processor, at any stage of a microinstruction's execution, needs only one of the four parts of the hardware. Hence the four processors share much of the same hardware because each is in a different stage of microinstruction execution. Such an arrangement, with several processors cycling through the same hardware, is often called a "barrel."

Figure 2 shows the data flow-paths of the four processors. Each processor, as mentioned above, has its own bank of 64 registers of 64 bits each; these are the only elements which are not shared among the processors. The functional elements, which operate on data, consist of the field extractor, which can extract any field of contiguous bits from a word, the field depositor, which can deposit an arbitrary length field into a word, and the arithmetic-logic unit (ALU), which can perform the usual arithmetic and logic operations plus a number of specialized ones. Note that the B-bus, which provides one of the two operands required by the ALU and field depositor, is fed by the field extractor, so that one of the operands could in fact come from any field of a word in a register.

An important issue which must be considered is whether or not the operations shown can be done with reasonable delay specifications, so that the processors are fairly fast. Calculations, using manufacturer's worst-case propagation delay specifications, show that a staging time of less than 100 ns, i.e., a clock rate of 10 MHz, is feasible. This yields a microinstruction execution-time per processor of 400 ns which (from the rule-of-thumb that 10 to 30 microinstructions are required for each target machine instruction) implies a target machine instruction execution time of 4 to 12 µs on each processor. The overall execution rate would be 107 microinstructions per second and be limited to about 0.3 to 1 million target machine instructions per second.

It is essential that the microinstruction set be both adaptable and powerful. To understand what this implies in terms of microinstructions, one must first realize what consumes most of the effort of emulators, namely, decoding the instructions of the target machine. In order to reduce this overhead, the set of microinstructions must include the means for extracting fields of bits from words and for making conditional branches and subroutine calls. All of these objectives are met with the CPU microinstructions chosen for CERF.

HARDWARE IMPLEMENTATION

CPUs

The Schottky TTL logic family was selected for implementation of the CPUs. Emitter-coupled logic (of the ECL 10K family) was considered but rejected because of power consumption, wiring (particularly the need for pull-downs), and interfacing problems (as it would require level translators to mate with other logic families), because of a relative shortage of MSI and LSI functions compared with TTL, and because of high cost. An ECL or ECL-TTL hybrid system could easily run twice as fast. Recent advances in bipolar memory technology not only reduced circuit complexity over preliminary estimates but also reduced system cost. Processor register banks are implemented with 64 X 9 RAM's (Fairchild 93419) which provide worst-case access time of 50 ns (actual measurements with a 1 GHz sampling system show an address-to-output access time of about 30 ns). Thus, the entire set of four banks of 64 X 64 registers requires only 32 integrated circuits. The microstore is constructed from 1024 X 1 bipolar RAM's (Fairchild 93415) with a worst-case access time of 70 ns (measurements show a typical access time of about 37 ns), requiring only 72 integrated circuits for a 1024-microinstruction storage.

The primary limiting factor in CPU speed will be delay in the ALU-test-branch decision path. Because branch options on every microinstruction are desirable and because branching should carry no time penalty (since it is done so frequently in microprograms), this path is crucial to our machine's performance. Our solution is to calculate both of the possible successors to each microinstruction and then select the correct one at the last possible instant.

Main Memory

Dynamic MOS RAM's were selected for main memory because of their low cost, availability, and reliability. Memory-system reliability is enhanced further by using an 8-bit Hamming code which provides single error-correction and double error-detection on each 64-bit datum. Thus, memory words are 72 bits in length.

Four-way interleaving is used to increase memory bandwidth which, for an approximate cycle time of 500 ns, is 8 megawords or 64 megabytes per second. Usable bandwidth, however, is somewhat less than this. New memory accesses are initiated while others are in progress for the purpose of minimizing idle time.

Selector Channels

Selector channels are essentially very fast 16-bit mini-computers and are implemented with 4-bit slice processor elements (AMD 2901). Each channel has 1024 words of bipolar memory (the same kind as the microstore) for programs and data. Special "vector" instructions are used to handle the extremely high data rates of the disc drives. Certain special functions, such as error correcting coding, are implemented in hardware.
RESEARCH GOALS

The main thrust of our project is the in-depth study of how a single operating system can effectively and efficiently coordinate the activities of more than one processor in a multiprocessor system. In such systems, each processor may have an independent address space mapped onto one physical memory or independent memories may be available to each processor. In either case, the operating system now must handle scheduling of a far more complex nature than on a uniprocessor system. Finally, a separate file and I/O system may exist for each processor. This results in yet another set of operations more complex than on conventional systems. Various problems may arise as a direct result of this type of system. Among these are:

1) Primary memory contention: since more than one simultaneously active processor and task can address the same memory module, techniques must be found to reduce or avoid memory contention at that module.

2) File protection: adequate file protection is difficult because active processors and tasks may simultaneously access the same files. File structures which insure data integrity must be developed and implemented.

A second key area of research on our project concerns the tradeoffs between computer architecture and operating systems. An operating system is very dependent upon system architecture (i.e., the instruction set, registers, etc.) of the processor on which it runs. This relationship is so close, in fact, that not only the effectiveness and efficiency of the operating system but the very structure of the operating system is determined by the processor architecture. For example, the “cactus stack” design of the B-6700 [Organick, 1973] leads naturally to the tree-like hierarchy of processes used by the operating system. Of course, one generally has to design an operating system for an unsuitable processor at the cost of decreased efficiency in conventional computer systems.

There are at least three basic areas in which the interaction between operating system and processor is crucial. These are processor scheduling, memory management, and file and I/O management. For each of these areas, tradeoffs can be made between performing important functions with “hardware macros” (e.g., with special processor instructions or using dedicated processors) or with “software macros” (i.e., aggregates of primitive hardware instructions). System speed can be increased by using more “hardware macros” in exchange for greater processor cost and complexity and perhaps reduced system flexibility. On the other hand, system cost can be decreased and system flexibility increased by using “software macros,” but at a reduction in system efficiency.

Existing systems are examples of a priori and ad hoc decisions regarding the distribution of functions between hardware and software. For example, essentially all operating system functions are implemented without hardware assistance in the IBM S/360 [IBM, 1970]. On the other hand, the Berkeley Computer Corporation BCC-500 [Wall, 1974] features specialized processors performing such functions as scheduling and memory management.

Our goal is to theoretically and empirically investigate several specific areas of tradeoffs between operating systems and computer architecture, using the CERF multiprocessor computer system as a laboratory tool. One such area is the design of operating systems for multiprocessors which emphasize synchronization primitives and the architectural features necessary to facilitate them. Hardware provisions for mutual exclusion are especially important in a multiprocessor system because the simple expedient of turning off interrupts and simulating mutual exclusion primitives in system software works only for a uniprocessor system. Since, for example, classic P and V operations may imply changes in processor scheduling, the relationship between the hardware and the software is very close at this point.

Another important area for research is the distribution of operating system functions to various system processors. The question here is whether important functions such as processor scheduling should be shared among the central processors, as in the original plan of the CMU Hydra system. Alternately, one of the central processors could be dedicated to these operations with the other processors acting as slaves. These functions could also be distributed to more specialized “peripheral processors,” as in the BCC-500 and the CDC 6000 and 7000 series. The CERF system is ideally suited for this type of research. Specifically, special microcode may be written for one of the central processors which will function as the master. Also, the drum controller may be reprogrammed to handle the memory management, and the Interdata 7/16 may be programmed to perform processor scheduling.

CONCLUSION

The design of CERF is presently complete, and we anticipate that its construction will be completed by June of 1977. This computer system has been designed as a laboratory tool in order to facilitate solution of the research problems described herein. In addition to providing a unique vehicle for research, CERF will also provide a flexible computer timesharing service to our engineering colleagues who are engaged in computer applications research. This user community, in fact, is necessary in order to provide typical user loads on the
system for the purpose of monitoring system performance.

Our initial target machine-architecture will be expandable so that we can add features as we need them with relative ease and so that well-structured, ALGOL-like languages can be fully supported and compiled efficiently. Because the basic host machine-architecture will remain relatively fixed, we will be able to generate meaningful performance comparisons as various increasingly complex computer systems are completed. We believe that such systematic studies with real systems will result in the data required for the design of computer systems of the future.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the support of the College of Engineering and Applied Science, the Department of Electrical Engineering, and the Naval Undersea Center (Contract #5-29037) in funding this project.

References


The Production Automation Project was one of the very first American projects launched, with funding from the RANN program of the National Science Foundation, to rectify the situation. The Project has been concerned since its inception with theories of mechanical manufacturing, design, and assembly, and with technologies and systems relevant to these areas. Its major goals are:

1) the development of modelling schemes to describe, in a geometrically complete manner, parts, stock (raw materials), assemblies, and the capabilities of particular tools;

2) the development of algorithms to produce automatically, from models (of parts, stock, and tools), manufacturing plans and command data for NC machine tools; and

3) design, implementation, and testing of integrated computer systems which embody such description and planning schemes.

Goals 1 and 2 are the intellectual keystones; in computer science terminology, they would be called problems in representation and planning.

Thus far our efforts have been directed mainly at theories and technologies for describing (representing, specifying) parts and assemblies, because it is pointless to study planning problems under Goal 2 before one can specify unambiguously the objects to be produced.

GEOMETRIC SPECIFICATION

The geometric aspects of part specification are critically important but poorly understood. Drafting practice implies that geometric specification should be viewed as a two- or three-phase process. Initially a nominal or ideal object—a "shape"—is defined, typically by an untoleranced drawing. In the second phase, dimensional tolerances are introduced; at this point one is no longer defining a single object, but rather a class of functionally equivalent objects. Attributes, which are conveyed by notes on engineering drawings, are specified in the final phase or in conjunction with tolerancing.

Engineering drawings are at best an imperfect medium for part specification. Because engineers and technicians possess vast stores of pertinent "world knowledge" (e.g., purpose of the device, general mechanical principles, etc.), they usually can extract from drawings the information needed to make and assemble parts correctly. Machines, e.g., programs for interpreting drawings, usually cannot. Thus new approaches are needed to the difficult problem of specifying precisely, to automatic manufacturing systems, what is to be made.

It is not hard to devise ad hoc schemes for manipulating geometry in computers. Indeed, the industrial world is becoming cluttered with systems which do just this, but these systems do not possess some basic properties that are essential for fully automatic production. For example, a reliable representation scheme should be complete and consistent; every part in some characterizable class should have a representation, and every representation should specify exactly one part. None of the
currently popular industrial graphics systems exhibit these properties [2].

To illustrate the issues, suppose that plane-faced polyhedra are represented by lists of their vertices. Does this representation scheme specify each polyhedron uniquely? Figure 1 shows by counter-example that it does not. Figure 2 shows that not even edge-list ("wireframe") representations are unique for polyhedra. (The construction of two different polyhedra having the edges shown in Figure 2 is left as an easy exercise for the reader.) Thus even for simple solids the problem of shape description is not trivial.

But there is more to geometric specification than shape description. As noted earlier, any industrially viable medium must provide means for specifying tolerances, surface finishes, and similar geometric attributes in a complete and consistent manner. Further, a viable system must be convenient for others (e.g., planning programs, humans) to use, and it must be reasonably efficient. We have devoted most of our effort since 1973 to the design, implementation, and testing of a medium which attempts to meet all of these criteria.

THE PADL SYSTEM

Our new medium for mechanical specification is a language called PADL (Part and Assembly Description Language) [3, 4]. It is oriented toward "simple," unsculptured mechanical parts, such as those shown in Figure 3, and it is based on the use of solid "building blocks."

The preparation of definitions in PADL follows the phases outlined above: shape specification, tolerancing, and attribute assignment. Shape specification is accomplished through constructive solid geometry, i.e., the definition of nominal objects as compositions of primitive solid objects via the operators listed in Figure 4. Figures 5-9 illustrate both the basic notion and the effects of particular operators. Figure 5 shows an "assembly" of two blocks. In the current version of PADL, assemblies are merely aggregates of distinct solids that can be referred to by a single name. Figure 6 illustrates the \texttt{INT.} (regularized set intersection) operator; it produces the volume common to the intersected objects and is analogous to arithmetic multiplication. Figures 7-9 show the effects of \texttt{UN.} (regularized set union) and \texttt{DIF.} (regularized set difference), which are analogous to arithmetic addition and subtraction. It is important to note that these operators can be applied to arbitrarily complex constructs; there are no "boundary conditions"
MECHANICAL SPECIFICATION: STAGE 1

FUNCTION PADL FACILITIES

SPECIFY NOMINAL 'SHAPE' OF PART
OR ASSEMBLY (CONSTRUCTIVE SOLID
GEOMETRY)

PRIMITIVES OPERATORS

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>MOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYLINDER</td>
<td>.INT.</td>
</tr>
<tr>
<td>(WEDGE)</td>
<td>.UN.</td>
</tr>
<tr>
<td>(CONE)</td>
<td>.DIF.</td>
</tr>
<tr>
<td>.ASB.</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4

Fig. 5

Fig. 6

Fig. 7

Fig. 8

Fig. 9

(to example, that objects to be merged via .UN. have coplanar faces) to be satisfied. Figures 10 and 11 are drawings produced automatically from PADL definitions of parts used in a Xerox copier.

To appreciate what remains to be done after a nominal shape has been specified, ponder the conventional drawing in Figure 12; it contains far more information than that required to specify a shape! Thus we proceed to Stage 2 of the definition process, which in PADL combines the second and third phases of geometric specification discussed earlier.

Figure 13 lists PADL's facilities for assigning dimensions, tolerances, and attributes [4, 5]. These are too elaborate to discuss here, but some feeling for their form and placement in PADL definitions can be garnered from Figure 14, which is a PADL definition for the part shown in Figure 12. The two-stage nature of the definitional process is evident in Figure 14. The PADL system uses the dimensioning and tolerancing information in PADL definitions to produce dimensioned drawings fully automatically (see Figure 11b).

Finally, Figure 15 lists our current objectives for the prototype PADL processor (with parentheses distinguishing desirable capabilities from those we are committed to achieving). Figure 16 depicts the logical organization of the prototype, and Figure 17 summarizes some relevant characteristics of its implementation. The prototype was designed for industrial portability, and we expect to
release it for public trials in 1977. It should prove useful as a medium for part description in its own right, or as an implementation vehicle for customized description systems.

SOME RESEARCH ISSUES

The PADL project and its precursor studies spawned a host of research topics, some of which we shall discuss briefly.

Modelling Manufacturing Activities

A distinguishing characteristic of discrete manufacturing systems is the critical role played by geometry, to which the ever-present blueprints attest. Indeed, the essence of discrete goods production is the application of geometric transformations to solid objects, i.e., geometric entities. Some of these transformations are the familiar rigid motions (translations and rotations) used in material handling and assembly. Other transformations, however, are much more violent and change the "structure" of objects. For example, sawing a piece of bar stock produces two objects having generally different shapes.

Any scheme for modelling discrete manufacturing systems obviously must offer powerful facilities for representing and manipulating geometric structures. The vector space methods which pervade engineering do not offer such facilities in any usefully explicit form, despite the fact that they have provided effective models for continuous flow processes. (Vector space models traffic in simple arrays of parameters called "state variables" in the terminology of Control Theory.) This subtle deficiency is, we suggest, one of the deep reasons why automation in the discrete industries is at a more rudimentary stage than in the continuous process industries.

Fortunately, powerful methods for manipulating symbolic structures are being developed in several branches of Computer Science [6], and our work both draws upon and contributes to this effort. Our special concern is the design and manipulation of symbolic representations
MECHANICAL SPECIFICATION: STAGE 2

FUNCTION PADL FACILITIES

SPECIFY TOLERANCES

&D-CHAIN

ATTRIBUTES NAMING FACILITIES

ASSIGNMENT FACILITIES

TOLERANCES

Conventional

'True Position'

Form

Relational

ATTRIBUTES

Finish

Threads

(Other)

DISCRETIONARY GEOMETRY

Bevels

Fillets

(Other)

Fig. 12

Fig. 13

Fig. 14

BEGIN(DEF(&PART3) )
BEGIN(NOMDEF(&PART3) )
&CYL 1 = $CZ(E,F) 
&CYL2 = MOV(&CYL 1) BY (-L,O,O) 
&LINK = $B(L,G,H) AT (-L,-N,I) 
&CAST = &CYL1 .UN. &CYL2 .UN. &LINK 
;

E = 1.5 
F = 1.5 
G = 0.5 
H = 1.0 
I = 0.25 
L = 3.0 
N = E/2 

&HOL 1 = $CZ(J,F) 
&HOL2 = MOV(&HOL 1) BY (-L,O,O) 
&PART3 = &CAST .DIF. (&HOL 1 .UN. &HOL2)

NOMDEF 

J = 0.75 : PM(.002,.000)

END( NOMDEF( &PART3) )

BEGIN(TOLDEF(&PART3) )

#A = ZFACE( , ,0)

#B = ZSURF(&CYL 1)

#C = ZSURF(&HOL 1)

###D = POINT(-L,-N,.750:B)

$$DA = DAT(#A)

$$DAB = DAT(#A,#B(MMC) )

$$DACD = DAT(#A,#C(MMC).###D) 

POSTOL(#C,$$DAB) = 0.014(MMC)

PERP(#C,$$DA) = 0.002(MMC) 

#K = ZSURF(&HOL2)

POSTOL(#K,$$DACD) = 0.007(MMC)

#M = ZFACE( , ,F)

PAR(#M,$$DA) = 0.005

TOL(F) = PM(.005)

TOL(L) = B

END(TOLDEF(&PART3) )

END(DEF(&PART3) )

BEGIN(DEF(&PART3) )
BEGIN(NOMDEF(&PART3) )

&CYL1 = $CZ(E,F) 
&CYL2 = MOV(&CYL1) BY (-L,0,0) 
&LINK = $B(L,G,H) AT (-L,-N,I) 
&CAST = &CYL1 .UN. &CYL2 .UN. &LINK 

END(DEF(&PART3) )
for geometrical and topological entities and transformations.

We shall illustrate some of the practical issues via PADL examples:

1) Detection of spatial interferences is important in planning tool motions to avoid collisions, in the design of assemblies, and for other purposes. Interference conditions can be tested easily in the PADL environment by a procedure which auto-

matically constructs "interference objects" according to the formula

\[ \text{INTOBJ} = \text{UN} \cdot (\text{OBJ}(i) \cdot \text{INT} \cdot \text{OBJ}(j)) \]

for all \( i, j \) where \( \text{UN} \) and \( \text{INT} \) denote regularized set union and intersection, respectively. Figure 18 provides an example: a poorly designed assembly whose parts do not fit. The (disconnected) interference solid for the two components of the assembly is shown in solid lines in Figure 18b.

2) Figure 19 illustrates the use of the set difference operator for the modelling of material removal operations. A cylindrical milling cutter moving on a straight-line trajectory sweeps the volume shown in Figure 19a. Figure 19b shows a block of "stock" superposed on the cutter-swept volume. Figure 19c depicts the result of the straight-
line milling operation, i.e., the difference of the two objects of Figure 19b. The ability to model removal operations as set differences is one of the primary considerations that led us to the concepts embodied in PADL.

PADL's usefulness as a geometric modelling tool is attributable to its use of solid primitive entities, and especially to the generality of its combinational operators. Thus, for example, we would be unable to detect the interference of two arbitrary parts defined in PADL if the set intersection operator were not applicable to arbitrary pairs of PADL-definable objects. PADL is the only modelling system known to us which has such general operators and is not restricted to (planar-face) polyhedra [7].

Foundations of PADL Geometry

Let us explore from a different viewpoint the notions of representational completeness and consistency discussed earlier through counter-examples. As a starting point we shall postulate, on pragmatic grounds, a property called "geometric completeness." An intuitive definition runs as follows:

"A specification is geometrically complete if it contains information sufficient to compute any geometric property of the specified entity."

Clearly, systems which admit only geometrically complete specifications solve the "geometry problem" once and for all (at least in principle). But can the notion of geometric completeness be captured precisely, in terms that admit formal verification for particular representation schemes?

Figure 20 suggests a useful way of looking at the problem. Firstly, objects in the real world must be replaced by mathematical models, i.e., by abstract entities defined in an appropriate domain of mathematics. In the usual 3-D Euclidean space \( \mathbb{E}^3 \) representation of the
physical world, subsets which are triangulable, compact, and regular furnish appropriate models for solid objects [8, 9, 10]. (This class of subsets excludes, for example, curves, surfaces, and other sets which are not "solid," i.e., homogeneously 3-D.) For brevity we shall call such sets "regular."

Next, we must define precisely the notion of "geometrical property." This is easily accomplished by defining geometric properties through mappings or functions which take regular sets (and perhaps other mathematical entities) and produce well-defined mathematical entities. Thus, "volume" is a geometrical property of an object because it can be defined through a function that maps regular sets into positive real numbers.

Let us now return to Figure 20. The mapping labeled "representation scheme" in the figure takes regular sets and produces mathematically defined symbol structures, e.g., strings. Eventually such symbol structures must be further mapped into storage structures in computers, but this step is not shown in the figure.

The notion of geometrical completeness may be expressed quite simply in this framework: it amounts to requiring that the representation-scheme mapping be invertible, i.e., that each representation correspond to a single and well-defined regular set. Clearly, if we know which set corresponds to a given representation, we have all the geometrical information required to determine geometrical properties (which we defined earlier as functions on a domain of regular sets).

Because a set is merely a collection of points, it can be defined uniquely through the following point membership function

\[ m(P,S) = \begin{cases} 1 & \text{if } P \in S, \\ 0 & \text{otherwise,} \end{cases} \]

wherein \( P \) is a point in \( E^3 \) and \( S \) is a subset of \( E^3 \). (The function \( m(P,S) \) is often called the "characteristic function" or "indicator" of the set.) Any representation that contains enough information to enable the point membership function to be computed also contains enough information to enable any other geometrical property of the object to be computed, i.e., the representation is geometrical complete. Thus the point membership function may be used in an obvious manner to provide a definition of geometric completeness which is equivalent to that presented above, and which is perhaps more appealing intuitively.

Finally, we must emphasize that we have discussed completeness only in the context of nominal or ideal-shape geometry. Variational completeness is equally important for dealing with tolerated objects, and it can be formalized in a rather similar manner. The PADL system is complete in both senses, i.e., nominally (shape geometry) and variationally (tolerance geometry).

Practical Applications of Object Representations

Representations of mechanical objects should be viewed for practical purposes merely as sources of data for procedures which compute useful geometric properties, e.g., volume, centroid, "appearance," and so forth. Our earlier discussion implies that all such procedures should be defined mathematically as mappings on a domain of regular sets.

Thus far we have studied only one practical application in detail: the graphic rendering of solid objects, i.e., computation of their "appearances." This work has two aspects: (1) the definition of "appearance" in terms of mathematical mappings from regular sets into 2-D "picture" or "drawing" spaces; and (2) the design of efficient algorithms that effect such mappings.

The most challenging problem we encountered in this work is boundary evaluation, i.e., conversion of a constructive geometry representation of a solid object into
a representation of its bounding surfaces. (This is necessary because the appearance of an opaque object depends on its topological boundary or frontier, not on its interior.) Our algorithms for boundary evaluation are the only ones in existence, insofar as we know, that deal with general (regularized) set-theoretical combinations of objects whose faces are not necessarily planar. They are based on a generalization of the membership testing ideas discussed previously but are too complex to describe here [11].

**Physical and Computational Complexity**

How does one design PADL-like languages to match the part populations found in industry? In particular, which primitive solids and operators should a language possess? We believe that such questions should be approached initially through the use of part surveys. This has led us—somewhat unexpectedly—into “ramblings in the experimental sciences of the artificial [i.e., man-made artifacts]” [12]. These ramblings raised questions such as the following:

1) How does one characterize part populations?
2) How is geometric complexity to be measured (defined)?
3) How is geometric complexity related to mechanical function?
4) How do geometric attributes of part populations influence the complexity of computations needed to evaluate geometric properties?

Figures 21 and 22 depict some of our attempts to relate the geometric complexity (measured crudely as the number of primitive instances in a PADL definition) of parts used in a Xerox copier to such characteristics as part cost and manufacturing process [13]. Work on the fourth question in the context of boundary evaluation is just beginning.

Virtually nothing is known about the sorts of questions raised above. This state of affairs may be due largely to the lack of powerful media in which to describe and characterize solid objects, and thus PADL’s role as a research tool may be as important as its (potential) industrial applications.

**CONCLUDING REMARKS**

By no means have we exhausted the list of research topics in this field, but we have said enough to convey their flavor. Single copies of project reports from the following list of publicly available reports will be sent on request.

**References**


Available Documents

TR-1-I Part I of a TECHNICAL REPORT—DISCRETE PART MANUFACTURING: Theory & Practice
by the Staff of the Production Automation Project

TR-1-A APPENDIX A of a TECHNICAL REPORT—DISCRETE PART MANUFACTURING: Theory & Practice
"SURVEY OF COMPUTER GRAPHICS"
by Jonathan E. Shopiro (with contributions by W. B. Fisher, H. B. Voelcker)

TR-1-D APPENDIX D of a TECHNICAL REPORT—DISCRETE PART MANUFACTURING: Theory & Practice
"SURVEY OF NUMERICAL CONTROLLER TECHNOLOGY"
by Alan E. Middleditch

TM-19 TECHNICAL MEMORANDUM 19 (currently in draft and under external review)
PART AND ASSEMBLY DESCRIPTION LANGUAGES—I
Dimensioning & Tolerancing Facilities in PADL-1.n
by A. A. G. Requicha
Anticipated Release: Autumn 1976

TM-20a TECHNICAL MEMORANDUM 20a
PART AND ASSEMBLY DESCRIPTION LANGUAGES—II
Proposed Specifications for Definitional Facilities in PADL-1.n & Tentative Specifications for Command Facilities
by A. A. G. Requicha, N. M. Samuel, H. B. Voelcker
August 30, 1974; Level "a" Revision: Nov. 30, 1974 ............................. 59 pp.

TM-21 TECHNICAL MEMORANDUM 21
METHODOLOGY AND RESULTS OF AN INDUSTRIAL PART SURVEY
by N. M. Samuel, A. A. G. Requicha, S. A. Elkind

TM-22 TECHNICAL MEMORANDUM 22
AN INTRODUCTION TO PADL: Characteristics, Status, and Rationale
by the Staff of the Production Automation Project
December 1974 ............................. 41 pp.
Reprinted June 1976

CRDM-5 COMPUTER RELATED DEVELOPMENT
MEMO 5
USERS' GUIDE TO STGPAK, A STRING PACKAGE FOR FORTRAN PROGRAMMERS, Version V001B
by W. Burns Fisher

CRDM-9 COMPUTER RELATED DEVELOPMENT
MEMO 9
GPAK: A SUITE OF FORTRAN SUBPROGRAMS FOR COMPUTATIONAL GEOMETRY & GRAPHICS
by W. A. Hunt, H. B. Voelcker

CRDM-10 COMPUTER RELATED DEVELOPMENT
MEMO 10
FLAP: A FLEXIBLE ARRAY PACKAGE FOR FORTRAN PROGRAMMERS
by N. K. Birrell, W. B. Fisher

CRDM-11 COMPUTER RELATED DEVELOPMENT
MEMO 11 (REPRINT)
VERSION 22 of the Oregon FLECS/FORTRAN PROGRAMMING SYSTEM
by Terry Beyer (Univ. of Oregon)
January 1975
AUTOMATIC DATA STRUCTURE SELECTION
Jerome Feldman, James Low, and Paul Rovner

One of the major weaknesses of current computing is inadequate associative memory. For a person, a word such as “Saigon” evokes a rich set of associations having marked effects on subsequent thoughts and perceptions. From the earliest days, computer scientists have attempted to understand such aspects of human associative memory and to develop similar capabilities in computer systems. Our work has concentrated on one small part of the problem: developing software associative memory systems for conventional digital computers.

Our first insight into the problem came in the early sixties when we noticed that the associative memory problem could be divided neatly into two subproblems: (1) languages for expressing associative processing; and (2) efficient ways of implementing such languages. Even if we had idealized associative memory computers, we would have the problem of expressing algorithms to be carried out by them. Starting from this point of view, we developed LEAP (Language for the Expression of Associative Processing) [Feldman & Rovner, 1969] and several versions of SAIL (not an acronym) [Van Lehn, 1973 and Feldman et al., 1972]. These systems have been fairly widely used and the basic ideas are now commonplace in the artificial intelligence languages [Bobrow & Raphael, 1974], relational data bases [Sibley, 1976], automatic programming [Balzer et al., 1974], etc. There is still a great deal to be done along these lines and we have begun work on a new language [Feldman, 1976] for expressing associative operations.

Of course no expressive mechanism will be useful for computing unless it can be implemented with some efficiency on available computers. It is this problem which has primarily concerned us, as well as many others, over the last dozen years. Before going into the specific aspects of efficiently implementing associative memory operations, we will present an example which will be used to illustrate various technical details as they arise. The example database is a collection of relations among family members, their genders, and their places of residence.

For our purposes, an associative memory can be viewed as a large collection of associations among items. For the most part, we will be concerned with three-part associations (triples) such as:

- Sexof · Abby = Female
- Parentof · Eric = Roberta
- or abstractly A · O = V. The A, O, V notation comes from considering the association as:
- Attributeof · Object = Value

In an associative memory, one could retrieve any such association by specifying any one or two of its positions. Table 1 describes these possibilities.

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A · O = V</td>
<td>Homeof · Paul = Valhalla</td>
<td>The association itself, if present</td>
</tr>
<tr>
<td>A · O = X</td>
<td>Parentof · Eric = PARENT</td>
<td>Parents of Eric</td>
</tr>
<tr>
<td>A · X = V</td>
<td>Parentof · CHILD = Paul</td>
<td>Children of Paul</td>
</tr>
<tr>
<td>X · O = V</td>
<td>RELATION · Eric = Rochester</td>
<td>Attributes linking Eric to Rochester</td>
</tr>
<tr>
<td>A · X = Z</td>
<td>Parentof · CHILD = PARENT</td>
<td>Child-Parent pairs present</td>
</tr>
<tr>
<td>X · Z = V</td>
<td>RELATION · PERSON = Male</td>
<td>All pairs of attributes and objects yielding Male</td>
</tr>
<tr>
<td>X · O = Z</td>
<td>RELATION · Abby = VALUE</td>
<td>All attributes of Abby and their values</td>
</tr>
</tbody>
</table>

29
An associative language will contain operations like:

Make Parentof • Eric = Paul

and

Erase Homeof • ANY = Valhalla

which add or delete associations or sets of associations from the collection. There will also be operations to retrieve and make use of associative information like:

Print (Sexof • Peggy)

and

Foreach PERSON such that

Homeof • PERSON = Rochester do

Print (PERSON, “lives in Rochester”).

In the latter case, “PERSON” is a variable that is repeatedly bound to the middle (Object) position of associations matching the pattern. We are interested in how to implement a programming system that can do all these operations efficiently.

There are many techniques available for implementing associations. Over several years we studied which representation or combination of representations would be “best” for implementing associative memory under various circumstances. Although it was very enjoyable and we learned a great deal from it, this quest for an optimal data structure was misguided. The difficulty is, of course, that each program is different and one cannot expect a single implementation to be best for all programs. This was known at the outset, but it is only recently that advances in several areas of computer science have enabled us to undertake the development of “intelligent compilers” which select a different collection of storage structures for each program. The first such effort [Low, 1976] was concerned primarily with sets and lists which are simpler than associations to deal with. Current efforts [Rovner, 1976] involve the automatic selection of representations for associations. We illustrate some of our work in this area in the rest of this section. The discussion will center around data structures which fit into some “main” memory. (The problem for structures in several different kinds of memory is more complex.)

The automatic selection system knows how to deal with a large number of different storage structures, but these are all based on a few fundamental concepts. One representation of a set of triples is simply to store triples in a vector of entries three items wide and to search all entries for answers to the questions in Table 1. There are a variety of record techniques in which a fixed location in a block of storage is associated with an attribute, e.g., one could have records for people with a Sexof field, etc. Another common technique is to form a linked list of attribute-value pairs for a given object. Other useful ideas include inverted lists (a link through all records containing some item), hash-coding [Morris, 1968], and using one-bit fields to represent the presence or absence of a property. We do not have space to discuss all of these, but we will give a typical example of how such representations are used by the current automatic selection system.

Let us consider the basic record-style representation of a triple. A typical choice for the sample universe described above would be to have a record for each person with fields for his Sex, Home, and Parents. Since there may be two (or more, west of the Rockies) parents, the Parentof field might point to a set of parents, yielding something like the following structure for Eric:

<table>
<thead>
<tr>
<th>Homeof</th>
<th>Rochester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sexof</td>
<td>Male</td>
</tr>
<tr>
<td>Parentof</td>
<td></td>
</tr>
</tbody>
</table>

There are additional complications, such as how to represent the set of parents. More to the point, this representation will not support answers to several of the possible associative searches such as:

Parentof • CHILD = Paul

without exhaustive search. Any selector of representations (automatic or not) must take into account all the ways in which a triple is used in the program. The automatic system contains a number of heuristic rules for deciding whether a given kind of representation is useful for a class of triples. These rules help narrow down the number of choices it considers. In addition, the system has detailed time and space formulas for estimating the cost of doing each possible operation on each possible storage structure type. For example, the space cost formula for a record representation for Parentof and the time cost formula for a search operation of the form

\[ A \cdot O = X \]

are shown in Figure 1. The space formula represents the requirement that there be a field of each record (0.5 cells) for each set-valued attribute (e.g., Parentof), and that space for the set of values is needed. In our example, there is only one attribute (Parentof), hence \( NA = 1 \). \( NO \) (number of objects) represents the number of people. \( NVALS(A, O) \) represents the average number of parents per person.

The flow chart for the search algorithm is also shown in Figure 1. The time cost formula represents the cost of the algorithm. \( C1, C2, \) and \( C3 \), are constant terms; to a good approximation they depend only on the machine and on basic design decisions and can be determined \( a \ priori \). \( P1 \) represents the probability that the search succeeds. The call on the SGENCOST function represents the execution time required to generate the elements of a set of a specified size \( (NVALS(A, O)) \). The values of \( P1, NO, \) and \( NVALS(A, O) \) are determined from answers to questions that the system asks. The choice of which questions to ask is based on which cost formulas are being used in a given run of the selection system.

The discussion below illustrates the behavior of the system using the following simple program as an example:
Sample Cost Formulas for Records

Space: \( NA \cdot NO^*(0.5 + SETSIZE (NVALS(A,O))) \)

Time: \( SEARCH (A \cdot O = X) \)
\( C_1 + C_2 + C_3 + P_1 \cdot SGENCOST (NVALS(A,O)) \)

Flow Chart for \( SEARCH (A \cdot O = X) \)

1. **C1**: locate the record for \( O \)
2. **C2**: locate the field for \( A \) within the record and pick up the pointer to the set of values
3. **C3**: is the set empty?
   - **no**: probability \( P_1 \)
   - **yes**: probability \( 1-P_1 \)
   - done

**SGENCOST(NVALS(A,O))**: generate the elements of the set and return them one at a time
   - done

---

**Figure 1**

BEGIN

Attributes Parentof, Sexof, Homeof
Item Constant Male, Female, Peggy, Abby, Rochester...
Item Variable CHILD, PARENT

MAKE Parentof \cdot Abby = Paul
MAKE Parentof \cdot Abby = Roberta
MAKE Sexof \cdot Abby = Female
MAKE Homeof \cdot Abby = Rochester
MAKE Homeof \cdot Thor = Valhalla

(1) Foreach CHILD, PARENT such that
   Sexof \cdot CHILD = Male
   and Parentof \cdot CHILD = PARENT
   do Print ("Sonof" PARENT "is" CHILD)

(2) Foreach CHILD, PARENT such that
   Parentof \cdot CHILD = PARENT
   and Homeof \cdot PARENT = Rochester
   do Print (CHILD "writes home to Rochester for money")

END

First let us consider the even simpler program with the part in the box omitted. It builds a collection of associations involving the three attributes. The "Foreach" part computes (parent, son) relationships in two steps. First it chooses an association of the form Sexof \cdot O = Male and binds CHILD to the value of "O" in the association. It then uses that value to match an association of the form Parentof \cdot CHILD = V and assigns the matching V to PARENT, and prints. This is repeated until there are no more matching associations. We will see how the current system chooses storage structures for this simple program.

In analyzing the simple program, the selector system first forms classes of associations. A class of associations is a group which will all be represented the same way internally. In this case, there are two classes: one based on the Parentof relation and one on the Sexof relation. Since there are no searches on the Homeof relation (with the boxed statement omitted), the compiler would not include those associations at all. For each class of associations, the system will characterize the operations performed on that class and the possible values for unbound variables. In our simple case, each class has several “MAKE” operations and one SEARCH. Because the SEARCH is different in the two cases, the system will suggest different storage schemes.

First consider the class based on Sexof. By keeping track of all the uses of the class, the system is able to determine that there is no use of associations with Sexof \cdot O = Female and these needn’t be represented. It then asks three questions about the use of the Sexof relationship. They are:

1. How many males are there likely to be? Sample answer: 5
2. How often will the Sexof search fail to match? Sample answer: Never
3. How often will a redundant MAKE occur? Sample answer: Never

Based on its analysis and on the answers to the questions, the system proposes and evaluates several possible storage structures. The best of these is a record having one field which points to the set of MALES (this set is represented as a linked list). Based on the cost formulas, this has an expected size of 5.5 cells and an expected time of 457 instructions. Another alternative would be to list each association separately in a table. This has an expected size of 13.0 and an expected time of 935. The set representation is better than this (and all others) in both time and space and is chosen. This is a very simplified (almost degenerate) case of the selection system’s operation.

In selecting a storage representation for the class based on the Parentof relation, the system goes through a similar procedure. In this case, it selects to assign a record for each person, with a field of the record indicating the set of parents.

Now let us consider the more complex program including the statement in the box. Since the Homeof re-
lation is used, there are now three classes of associations.
In our tiny sample program, the only use of Homeof is to
test if someone lives in Rochester. This, plus the fact that
there is a small fixed number of people, causes the selec­
tion system to choose as the best structure a single word
per person indicating whether he lives in Rochester. This
representation is estimated to require one cell and 300
instructions.

A more interesting effect of adding the boxed state­
ment is to change the choice for representing the class
based on the Parentof relation. Now the program calls
for enumerating all Child-Parent pairs as well as finding
the PARENT of a given child. The selection system con­
siders making two separate data structures or making
one data structure which does both jobs. The best choice
is to have a list of all children with each child indicating
the set of his parents. This is estimated to require 16.5
cells and 1600 instructions.

The problem of automatically selecting representa­
tions for higher level data structures is much more com­
plex than this discussion indicates. Some of the addi­
tional problems are: dealing with sharing and redund­
dancy in storage structures, representing the symmetries
of associative operations and storage structures effi­
ciently, choosing a level of detail for the cost formulas
that makes the problem tractable, and developing meth­
ods of program flow analysis to identify the different
high-level data structures of the program and character­
ize how they are used.

A representation-selection system should also consider
the possibility of representing the same relation in dif­
f erent ways in different parts of the program. Often a
program consists of several distinct phases in which the
types of access to the relation differ greatly. The best
choice may be to allow different representations in the
different phases with a translation between the repre­
sentations where a new phase is begun.

It is important to realize that the attributes of data
sets for programs may change greatly over time. It is
therefore necessary to include monitoring in completed
programs so that we may determine whether we should
alter our choice of representation. An important area of
research is to determine how to efficiently obtain the
information which would allow us to intelligently change
the representation of data structures.

One of the hardest questions to answer about data
representation is whether a particular data structure
should be represented explicitly or not. For example, a
program making a large use of the function SINE would
benefit from having a table of argument-value pairs. A
program using SINE infrequently would be better off
recomputing the value of SINE each time it needs the
value rather than wasting storage with a table for the
function.

Data structures on secondary storage, which have a
lifetime greater than the execution of a sample program,
present another set of difficulties. Often there is not one

single program which accesses and updates the data
structure but many such programs. To optimize over
many programs (some of which may not yet be written)
is extremely difficult. The costs of changing representa­
tion for data structures in secondary storage are much
higher, and so we must be very careful to choose appro­
priate representations the first time.

There are also program transformations which can
greatly improve the overall performance of the system.
For example, let us look again at the boxed statement.
An equivalent statement would reverse the two searches
giving:

\[ \text{Homeof} \cdot \text{PARENT} = \text{Rochester} \quad \text{and} \quad \text{Parentof} \cdot \text{CHILD} = \text{PARENT} \]

This uses the associations very differently, first choosing
people who live in Rochester and then choosing children
of only these parents. For large data sets, this can be
much more efficient than the original statement. One
would like an intelligent compiler to make these im­
provements as well. There has been a little work on this
subject [Hilbing, 1969] but it has only scratched the sur­
face of this intriguing and important problem.
References
SEMINAR MEETINGS

Fall, 1974
Robert Sproull, Xerox Palo Alto Research Center
"Computer Graphics: Hidden Line Elimination"
Pierluigi Della Vigna, Istituto di Elettrotecnica ed Elettronica (Milano)
"Identification of Syntax-Directed-Translation Schemes"
Richard E. Fikes, Stanford Research Institute
"Modeling and Planning Mechanisms for a Computerized Consultant"
Mark Gold, University of Montreal
"Hard and Easy Cases of the Deadlock Problem"
Roger Schank, Yale University
"Computer Understanding of Natural Language"
Lee D. Erman, Carnegie-Mellon University
"The HEARSAY Speech Understanding System and Why This is Computer Science"
Lawrence Landweber, University of Wisconsin
"Axiomatic Equivalence of Straight Line Programs with Structured Variables"
Herbert B. Voelcker, University of Rochester
"Production Automation Project"
Irwin Sobel, Columbia University
"An Application of Computer Vision to Biological Image Processing"
Bruce G. Buchanan, Stanford University
"Scientific Theory Formation by Computer"
Neil Wilheim, University of Rochester
"Stochastic Models for Computer Systems"
Robert Waag, University of Rochester
"Digital Processing and Display of Cardiac Ultrasound Data"

Spring, 1975
Gerald J. Sussman, Massachusetts Institute of Technology
"The Place of Teleology in Reasoning about Deliberate Artifacts such as Programs and Circuits"
Daniel Lehmann, Brown University
"Algebraic Structures for Paths Algorithms"
Raymond Reiter, University of British Columbia
"A Semantically Guided Deductive System for Formal Inferencing"
Daniel J. Rosenkrantz, General Electric Company, Research and Development Center
"Approximate Algorithms for the Traveling Salesperson Problem"
Charles Rieger, University of Maryland
"CONCEPTUAL OVERLAYS: A Paradigm for the Interpretation of Sentence Meaning in Context"
Charles Prenner, Harvard University
"The Implementation of Control Structures for Programming Languages"
Ronald L. Rivest, Massachusetts Institute of Technology
"A Generalization and Proof of the Aanderaa-Rosenberg Conjecture"
Jin Kue Wong, Cornell University
"Planar Graph Isomorphism"
Donald W. Loveland, Duke University
"Mechanical Theorem Proving: Updating the Problem Reduction Method"
Ron Baecker, University of Toronto
"On Program Illustration"
Susan S. Owicki, Cornell University
"Axiomatic Proof Techniques for Parallel Programs"
Derek C. Oppen, University of Toronto
"On Logic and Program Verification"
William Martin, Massachusetts Institute of Technology
"Towards an Automated Business Consultant"
Gerald M. Belpaire, University of Wisconsin
"Towards a Synthesis of Synchronization Problems"
Christopher M. Brown, University of Edinburgh
"The Edinburgh Robot Project"
Teofilo F. Gonzalez-Arce, University of Minnesota
"On Finding Approximate Solutions to Some Problems"
Eugene C. Freuder, Massachusetts Institute of Technology
"A Computer System for Visual Recognition Using Active Knowledge"
James Horning, University of Toronto
"Language Design for Reliability"

David Zaucha, University of Rochester
"Detection of Abnormalities in Brain Scans by Computer"

*One-week Visiting Professors*
Anil Jain, SUNY at Buffalo
“Multi-Dimensional Techniques in Computer Image Processing”

Robert Morris, Carleton University
“Minicomputers/Graphics Processors in Education and Research”

Ludwig Braun, SUNY at Stony Brook
“Computer Simulations in the Classroom”

LTC. Alan Salisbury, U.S. Army Tactical Data Systems
“Microprogramming: Yesterday, Today, and Tomorrow”

Fall, 1975

Dana H. Ballard, University of Rochester
“Hierarchic Recognition of Tumors in Chest Radiographs”

Oded Kariv, The Weizmann Institute of Science (Rehovot)
“An O(N^2.5) Algorithm for Finding Maximum Matching in a General Graph”

Edward S. Angel, University of Rochester
“Image Processing and Finite Difference Methods”

Harry E. Pople, Jr., University of Pittsburgh
“Dialog—A Model of Diagnostic Logic for Internal Medicine”

Nathan Friedman, University of Toronto
“The Multiplicative Complexity of Numerically Stable Algorithms for the Multiplication of Scaled Matrices”

Azriel Rosenfeld, University of Maryland
“Some Basic Image Segmentation Techniques”

Yorick Wilks, University of Edinburgh
“Frames, Scripts, Stories and Fantasies”

Allan Borodin, Cornell University (on leave from University of Toronto)
“Unifying Themes in Computational Complexity and Mathematical Challenges in Theoretical Computer Science”

Jill Meekesh, Yale University
“Using Planning Structures to Generate Stories”

Richard J. Lipton, Yale University
“On Vector Addition Systems and Reachability Problems”

Jan P. Allebach, Princeton University
“Digital-Optical Signal Processing”

Spring, 1976

John Seely Brown, Bolt Beranek and Newman Inc.
“The Use of AI Techniques to Create ‘Intelligent’ Instructional Systems: or How Things that Appear Simple—Often Aren’t”

Donald Michie, University of Edinburgh
“Advice Theory: A Numerical Method for Measuring Knowledge”

Theodore H. Kehl, University of Washington
“LM^2—BASIL—A HLL Minicomputer”

Robert C. Waag, University of Rochester
“Computer-Based Ultrasonic Determination of Tissue Structure and Imaging of Heart Motion”

David Pessell, Edward Angel, Charles Hohler and Harold Fox, University of Rochester
“Computers in Obstetrics”

Gary L. Miller, University of Waterloo
“Riemann’s Hypothesis, Tests for Primality, and the P-NP Question”

Thomas D. Howell, Cornell University
“Bilinear Forms and Tensor Rank”

David Marr, Massachusetts Institute of Technology
“Processing Visual Information”

Josef Hatvany, Hungarian Academy of Sciences
“Computer-Aided Design and Computer-Aided Manufacturing”

Daniel D. McCracken, Author and Consultant
“Where Computers and Conscience Meet”

Thomas G. Szymanski, Princeton University
“Evaluating Relational Expressions”

Dana Angluin, University of California at Berkeley
“The Complexity of Inference of Regular Languages”

Jonathan Broido, University of Rochester
“Inductive Machines: Some Decidability Problems”

Gabor T. Herman, Mayo Clinic and SUNY at Buffalo
“Rapid Computed Tomography”

Mitchell Marcus, Massachusetts Institute of Technology
“Diagnosis as a Notion of Grammar”

C. Beeri, University of Toronto
“A Synthesis Algorithm for Relational Data Base Schemas”

David C. Walden, Bolt Beranek and Newman Inc.
“Seminar on the ARPANET and Related Developments”

Gary D. Knott, Department of Health, Education, and Welfare
“A Talk on Storage and Retrieval Algorithms”

Scott E. Fahlman, Massachusetts Institute of Technology
“The Logic of Program Synthesis”

Richard Waldinger, SRI
“A System for Representing and Using Real-World Knowledge”

Richard Waldinger, SRI
“The Logic of Program Synthesis”

David Bobrow, Xerox Palo Alto Research Center
“New Results in Natural Language Understanding Systems”

Neil Wilhelm, University of Rochester
“An Overview of CERF”

Gary L. Miller, University of Waterloo
“Riemann’s Hypothesis, Tests for Primality, and the P-NP Question”

Thomas D. Howell, Cornell University
“Bilinear Forms and Tensor Rank”

David Marr, Massachusetts Institute of Technology
“Processing Visual Information”

Josef Hatvany, Hungarian Academy of Sciences
“Computer-Aided Design and Computer-Aided Manufacturing”

Daniel D. McCracken, Author and Consultant
“Where Computers and Conscience Meet”

Thomas G. Szymanski, Princeton University
“Evaluating Relational Expressions”

Dana Angluin, University of California at Berkeley
“The Complexity of Inference of Regular Languages”

Jonathan Broido, University of Rochester
“Inductive Machines: Some Decidability Problems”

Gabor T. Herman, Mayo Clinic and SUNY at Buffalo
“Rapid Computed Tomography”

Mitchell Marcus, Massachusetts Institute of Technology
“Diagnosis as a Notion of Grammar”

C. Beeri, University of Toronto
“A Synthesis Algorithm for Relational Data Base Schemas”

David C. Walden, Bolt Beranek and Newman Inc.
“Seminar on the ARPANET and Related Developments”

Gary D. Knott, Department of Health, Education, and Welfare
“A Talk on Storage and Retrieval Algorithms”

Scott E. Fahlman, Massachusetts Institute of Technology
“The Logic of Program Synthesis”

Richard Waldinger, SRI
“A System for Representing and Using Real-World Knowledge”

Richard Waldinger, SRI
“The Logic of Program Synthesis”

David Bobrow, Xerox Palo Alto Research Center
“New Results in Natural Language Understanding Systems”

Neil Wilhelm, University of Rochester
“An Overview of CERF”

David Pine, West Virginia University
“Some Hardware and Software Design Considerations for Data Base Management Systems on Associative Processing Systems”

Walter A. Burkhard, University of California at San Diego
“Partial-Match Queries and File Designs”

*One-week Visiting Professors*
David Zipser, Cold Spring Harbor Laboratory
"How Genes Build Brains"
Hanan Samet, University of Maryland
"Automatically Proving the Correctness of Translations Involving Optimized Code"
Alan Gary Nemeth, Harvard University
"Random Variables as a Data Type"
Kenneth Iverson, IBM Corporation
"APL as a Mathematical Notation"
Deepak K. Goyal, Princeton University
"Analysis of Parallel Scheduling Algorithms"
James R. McGraw, Cornell University
"A Systems Design Language for the Design of Operating Systems"
PUBLICATIONS
1975–76

Books and Chapters

Refereed Papers
Brown, C. M., “Principal Axes and Best-Fit Planes, with Applications,” Proceedings of MSAC’76, April, 1976.

Technical Reports and Conference Papers

37
GRANT SUPPORT


National Science Foundation, Production Automation Project, June, 1972 to February, 1976.

