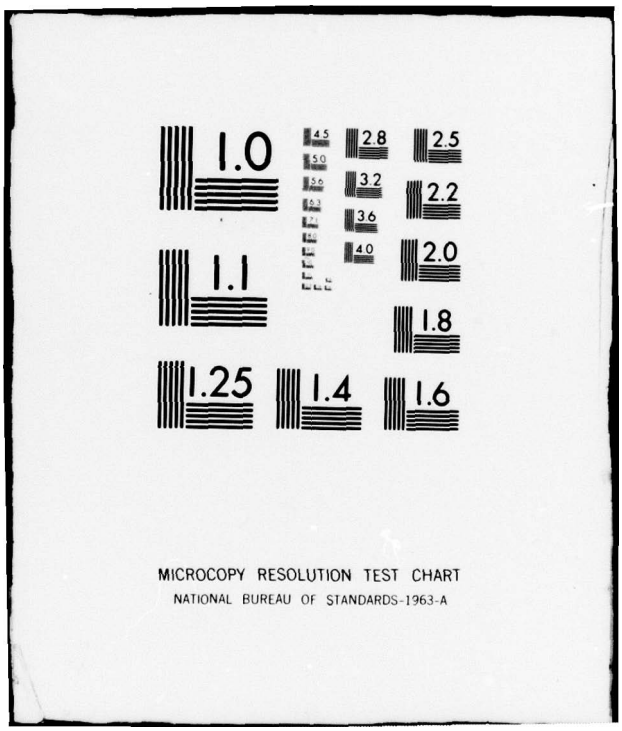
AD-A079 709 WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING.(U)
OCT 79 R P COOK
UNCLASSIFIED MRC-TSR-2008 NL

1 OF 1
AD-
A079709

END
DATE
FILMED
2-80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A079709

(9) MRC Technical Summary Report # 2008

(6) *MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING,

(3) B.S.

(10) Robert P. Cook

LEVEL II

(14) MRC-TSR-2008

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

DDC
RECEIVED
JAN 23 1980
E

DDC FILE COPY

(11) Oct[redacted]79

(12) 31

(Received August 20, 1979)

Approved for public release
Distribution unlimited

Sponsored by

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709

80 1 15 052
221 200

UNIVERSITY OF WISCONSIN-MADISON
MATHEMATICS RESEARCH CENTER

*MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING

Robert P. Cook

D D C
RECEIVED
JAN 23 1980
E

Technical Summary Report #2008
October 1979

ABSTRACT

Distributed programming is characterized by high communications costs and the absence of shared variables and procedures as synchronization tools. *MOD is a high-level language system which attempts to address these problems by creating an environment conducive to efficient and reliable network software construction. The concept of a processor module is introduced as well as a methodology for distributed data abstraction and process communication. In addition, a VHLN (virtual, high-level language network) is proposed for system development.

AMS(MOS) Subject Classification: 68A05, 68A10, 68A55

Key Words: distributed programming, modula, processor module, data abstraction, programming languages

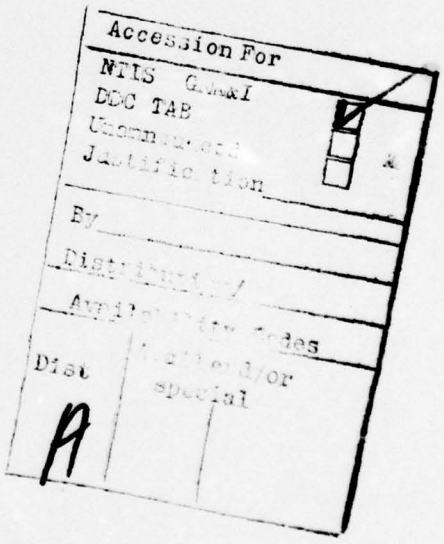
Work Unit #8 (Computer Science)

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024 and the Computer Sciences Department, University of Wisconsin, Madison.

This document has been approved for public release and sale; its distribution is unlimited.

SIGNIFICANCE AND EXPLANATION

A language (*MOD) for distributed programming is being designed and implemented at the University of Wisconsin to facilitate research in network software concepts. Distributed programming is characterized by the use of multiple hardware processors to implement an algorithm. The *MOD system also proposes a convenient methodology for the debugging and development of distributed programs. In addition, the language contains some new approaches in the areas of data abstraction, mutual exclusion, and synchronization.



The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the author of this report.

*MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING

Robert P. Cook

1. Introduction

*MOD(starmod) is a language, derived from Modula[27], which is intended for systems programming in the network environment. The *MOD project is based on experience with our PDP11/VAX Modula compiler[4] and was inspired by Brinch Hansen's "distributed processes" concepts[3]. The design strives to address the systems programmer's traditional concern for efficiency and includes the constraint that each language feature should be maximally extensible. For example, the *MOD user can utilize the data abstraction mechanisms to construct either queue or stack types; thus, the language attempts to define an appropriate set of primitives which can be extended to meet programming needs. By giving each user the freedom to experiment with language constructs for distributed programming, *MOD is also intended as a mechanism for research. This paper discusses the rationale behind the design of the *MOD system and contrasts the language features chosen with those of the Department of Defense(DoD) ADA language[15], Hoare's Communicating Sequential Processes (CSP)[14], Feldman's PLITS[2,9], and Brinch Hansen's "distributed processes"[3]. In particular, we address the distributed programming problem areas of interprocessor communication, type checking, separate compilation, debugging, and kernel efficiency. The *MOD language definition[5] should be consulted for the details of design decisions in other areas such as data abstraction or synchronization.

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024 and the Computer Sciences Department University of Wisconsin, Madison.

2. System Overview

Before proceeding further with a more detailed discussion of the distributed programming features, we will consider the module concept of Modula as a focal point for program development. A module encapsulates an environment and defines the relationship between itself and the outside world; therefore, both the information-hiding properties proposed by Parnas[20] and the flexibility of the Simula[6] "class" mechanism are maintained. Each module usually corresponds to a program abstraction and consists of an external interface specification, data structure definitions, procedures, processes, and an optional initialization part.

A computer system is traditionally[25] viewed as a collection of processors, processes, and procedures. A processor executes commands or instructions, a procedure is a sequence of instructions for a processor, and a process is one or more procedures together with the information necessary to control and to define the virtual processor on which it runs. *MOD provides these entities in the forms of a "processor module", procedure and process declarations, respectively. In addition, a "network module" is required to define system connectivity for the processors and to declare any global types or constants. These module types can be declared with the following syntax:

```
MODULEDECLARATION:=

        MODULETYPE;  [external; ]
        [define ELEMENT [,ELEMENT ]...; ]
        [export ELEMENT [,ELEMENT ]...; ]
        [pervasive ELEMENT [,ELEMENT ]...;]
        [BLOCK ]
        end IDENTIFIER

MODULETYPE:=

[interface | processor ] module IDENTIFIER|

network module IDENTIFIER=LINK [,LINK ]...

BLOCK:=

   [import IDENTIFIER [,IDENTIFIER ]...; ]
   [DECLARATIONLIST  ]
   begin STATEMENTLIST

LINK:=    (PROCESSORID [,PROCESSORID ]...)

ELEMENT:= IDENTIFIER [(PROCEDUREID) ]
```

The IDENTIFIER names the module and must be matched by the IDENTIFIER at the end of the BLOCK. The BLOCK consists of declarations for constants, types, variables, modules, processes, or procedures as well as a STATEMENTLIST which can be used to initialize the module. The module boundary delineates a closed lexical scope which can only be superseded by the explicit specification of "define", "export", or "import" lists.

An IDENTIFIER specified in an "import" list causes a declaration from a global scope to be made accessible within the module. The "export" attribute allows a local declaration to be visible at the enclosing lexical level; while "pervasive" makes the IDENTIFIER known at all lexical levels where the same name is not already declared. The latter option is most useful for system-wide constant and type definitions. The optional PROCEDUREID can be used to specify automatic initialization for exported types. The "define" statement is provided as an alternative to "export". It gives the user the ability to list those IDENTIFIERs which can be referenced externally, but only by prefixing the reference with the module name as with the Simula "class" notation. Furthermore, the "define", "export", and "pervasive" statements provide implicit read-only protection for any variable so listed. The ability to specify the external interface for each module is becoming a standard feature of modern programming as is demonstrated by its use in Mesa[10], Euclid[13], Alphard[24], ADA[15], etc.

Each processor LINK specifies a list of processor modules which can be sent messages. No variables or shared code are allowed at the network level; any procedures used for type implementations are replicated in the appropriate "processor module"s. Each "processor module" can represent any number of physical processors as long as they all use a shared memory for instruction execution. The external interface specification for a "processor module" lists any message types and process names which are used for communication. We should also point out that the availability of a hardware multiprocessor to implement a particular "processor module" should be regarded as a fortuitous circumstance

and should not be counted on by the programmer. An "interface module" is a Modula construct[27], similar to a monitor[13], which guarantees mutual exclusion across all the contained procedures.

The example in Figure 1 illustrates these concepts with a ring network version of Dijkstra's Dining Philosophers[7] problem. As in the original version, five philosophers are each trying to eat a plate of special spaghetti which has been placed in the middle of a round table. In our example, each philosopher can only directly control the right-hand fork; to get the left fork, the philosopher to the left must be consulted. However, each philosopher is also restricted to conversation with the right neighbor only; therefore, a message must be sent around the ring(table) to get permission to use the left-hand fork and to give it back. The algorithm is based on an ordered resource allocation strategy developed by Havender[12] which prevents deadlock and starvation.

The "diningroom" network definition specifies the connectivity for the ring network, defines a "semaphore" data type, and contains the program stubs for the five philosophers. Since "semaphore"s are declared as "pervasive", they will appear as builtin types in all processors in which "semaphore" is not redeclared. The five processors are specified only in terms of their external interfaces (termed a program stub). The keyword "external" indicates that the processor definitions are part of a separate compilation. The binding process for separate compilations will be discussed in Section 4.

Figure 1

```
network module diningroom=(phil0,phil1),(phil1,phil2),(phil2,phil3),
                          (phil3,phil4),(phil4,phil0); (*ring network*)
    interface module semaphoredef;       (*Boolean semaphore abstraction*)
        pervasive semaphore(init),P,V; (*available to all processors*)
        type semaphore = record taken: boolean;
                                free : signal;
                                end record;
        procedure P(var s: semaphore);
            begin
            if s.taken then wait(s.free) end if;
            s.taken:=true
            end P;
        procedure V(var s: semaphore);
            begin
            s.taken:=false;  send(s.free)
            end V;
        procedure init(var s: semaphore);
            begin s.taken:=false
            end init;
        end semaphoredef;
    processor module phil0;  external;
        define get,put,got;              (*program stub for philosopher zero
        process get(who,fork: integer);  (*get "fork" for "who" *)
        process put(fork: integer);      (*give "fork" back     *)
        process got(who: integer);       (*tell "who" the news  *)
        (*referenced externally as phil0.get, etc. *)
        end phil0;
        :
    processor module phil4;  external;
        :
        end phil4;
    end diningroom.
```

3. Language Concepts

From the *MOD viewpoint, a computer network can be characterized as an arbitrary collection of processors with fixed communication paths for interprocessor message transfer. Messages are assumed to range from no content(signal or interrupt) to arbitrary data structures. Furthermore, we require strong type checking both within and across processors to maintain system consistency. Finally, any mechanisms presented should be efficient and should not constrain the options of the systems programmer. For these reasons, we developed a process-oriented communication methodology which eliminated the need for additional statements to handle messages. In the next sections, the *MOD design will be presented along with a detailed discussion of the alternatives, advantages and disadvantages.

3.1 Processes and Signals

Each "processor module" consists of one or more concurrent processes declared as follows:

```
PROCESSDECLARATION:=

        process IDENTIFIER [(FORMALS) ]
             [ '['EXPRESSION']' ] [:TYPEID];
            BLOCK
            end IDENTIFIER

PROCESSREFERENCE:=

        PROCESSID [(ARGUMENTLIST) ]
```

Except for the keyword "process" and the optional priority EXPRESSION, the declaration is identical to that of a procedure; however, the semantics are different since a process can execute independently of its creator. Each instance of a process is created by a PROCESSREFERENCE which must specify a list of arguments corresponding exactly in type and number to the FORMALS. At this point, storage space is obtained for the activation record and the process control block, both of which remain allocated until the process terminates. The returned value for a functional process is set by assignment to the process identifier and must match the specified TYPEID. When a functional process exits, the returned value is copied from its activation record to the address space of the caller. The use of a functional process corresponds to sending a message and then waiting for a reply while a reference to a non-functional process implies parallel execution. We will frequently use the term message as a synonym for the record containing the arguments to or result from a process.

The optional EXPRESSION must evaluate to a compile-time constant which specifies the initial(default zero) priority of the process. Each process' priority can be modified by assignment to the variable "priority" which is used to control context switches among processes. The general rule is that a process loses control of the hardware processor if it lowers its priority below that of another "ready" process or if a higher priority process changes to the "ready" state. The other builtin process identifier("origin") is a time stamp which indicates the creation order relative to all other processes in the same processor

module. The example in Figure 2 completes the Dining Philosophers network by defining the actions of each processor and serves as an illustration of the preceding definitions.

Each philosopher is required to request the forks in a specific order and must have obtained the first fork before requesting the second. The "get" process accepts fork requests and either passes the request to the right in the ring or else gets control of the fork and sends an acknowledgment to the "who" philosopher. The "got" process uses its higher priority to speed the acknowledgments to the appropriate philosophers. Finally, it should be noted that multiple activations of each process can coexist; for instance, three independent copies of the "get" process could be handling requests simultaneously.

The signal construct embodies a message capability that is even simpler than a process call in that its arrival represents the only content. "signal" can be used as a basic type in *MOD to declare variables which can only be manipulated by the following procedures:

send(SIGNALID) wait(SIGNALID,RANK)

awaited(SIGNALID)

The interpretation of these procedures is identical both within and across processors. A "wait" delays the executing process in a local priority queue specific to SIGNALID. The queue is ordered first by the RANK attribute and secondly by the longest wait time. The "awaited" function returns a Boolean value which reflects the status of the "wait" queue(true=not empty) for SIGNALID. The "send" statement unlinks the process at the head of SIGNALID's queue and sets the process' status to "ready". If no

Figure 2

```
processor module phil_i;    (*0≤i≤4*)
    define get,put,got;          (*n=i+1 mod 5, right neighbor*)
    import phil_n;
    const first=min(i,i+4 mod 5);
          second=max(i,i+4 mod 5);
    var   myfork,gotit: semaphore;
    process get(who,fork: integer); (*get fork for who*)
         begin                      (*calls who on success*)
         if fork <> i then phil_n.get(who,fork);
                      else P(myfork); got(who);
                      end if;
         end get;
    process put(fork: integer);     (*give "fork" back*)
         begin              (*wake up anyone waiting*)
         if fork <> i then phil_n.put(fork);
                      else V(myfork);
                      end if;
         end put;
    process got(who: integer) [1];   (*let "who" use "fork*)
         begin
         if who <> i then phil_n.got(who);
                     else V(gotit);
                     end if;
         end got;
    procedure getfork(fork: integer); (*philosopher waits for "fork"*)
         begin get(i,fork);  P(gotit);
         end getfork;
     begin loop
           (*think*) getfork(first); getfork(second);
           (*eat*)   put(first); put(second);
           end loop;
     end phil_i.
```

process is waiting for the signal, the "send" is ignored. The semaphore example in Figure 1 uses both the message and delay capabilities of signals to build a synchronization primitive.

For interprocessor communication, the "import" processor can only perform "send" operations on a signal while the "export/define" processor is unrestricted. Thus, a signal can be thought of as a means of generating a name for a processor which also embodies a communication capability. The capability is also revocable in the sense that if the defining processor never "wait"s for the signal, the signal will always be ignored by definition.

3.2 Design Decisions

One of the major differences of opinion in distributed programming language design occurs over the use of process-oriented versus message-oriented communication. The DoD ADA language is a typical example of the latter choice. In a recent paper by Lauer and Needham[19], they state "that these two categories are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other." We agree with the duality conclusion but feel that the message-oriented approach has some deficiencies for our purposes; these deficiencies may be completely irrelevant in other applications. The following example from ADA[16] will illustrate most of these points.

```
task SEMAPHORE is     task body SEMAPHORE is
    entry P;                begin loop
    entry V;                        accept P;
end;                                accept V;
                              end loop;
                        end;
            :
            initiate SEMAPHORE;
            :
            P;
            <critical section>
            V;
```

The "initiate" statement causes the creation of the SEMAPHORE task which is equivalent to a *MOD process; note that a second "initiate" while SEMAPHORE is still active would be an error. The "accept" statement defines a message reception point within a task as follows:

```
accept ENTRYNAME [(FORMALS) ]
    [do STATEMENTS
 end [ENTRYNAME ]   ];
```

The procedure call syntax is used to send a message to an "accept" point. The "accept" statement must be executed by an active task to receive a message. The execution semantics are as follows. "Whichever(receiver or sender task) gets there first waits for the other. When the rendezvous is achieved, the appropriate parameters of the caller are passed to the called task ... The caller is then temporarily suspended until the called

task completes the statements embraced by do ... end. Any out parameters are then passed back to the caller and finally both tasks proceed independently of each other."[16]

Since "accept" is an executable statement, the receiving task can choose the execution point at which to receive the message; thus, mutual exclusion among competing messages is provided automatically. Also, the receiving task(process) remains static while a *MOD process is activated for each message. Other useful properties of the message-oriented approach are listed in Lauer and Needham[19]. However in our opinion, the message-oriented approach has the following disadvantages for a systems programming language.

Message Queuing. Since the receiving task in a message system can only process one message at a time, the kernel must queue any additional messages which arrive before the next "accept" statement is executed. In *MOD, message queuing, stacking, etc. are choices under user control since every message has a process to implement the delay protocol. Several examples of the *MOD/Modula programming approach may be found in Wirth[28,30] or in an application description by Andrews[1].

Active Processes. In *MOD, the recipient of a message is normally passive(no activation record or process control block); in the message-oriented approach, the task is always active although it may be delayed at an "accept" statement. Consider the semaphore example. Every semaphore used in an operating system would be an independent task with an activation record and control block. This fault is corrected in *MOD by separating message transmission and synchronization facilities. In addition, the *MOD user

can always create a static process but the user of a message-oriented system can never create a passive process which can "accept" a message.

Delayed Processes. The caller is always suspended until the called task completes the entry routine. This defect is present in both ADA and "distributed processes" but not in PLITS, *MOD, or Lauer and Needham. The primary reason for delay is to wait for a returned message. *MOD provides all combinations of "delay until called process terminates" and "delay until called process returns a message" except for "no delay but message returned" which can be easily programmed.

Process Deletion. It is dangerous to delete processes because of possible late arrivals or of the potential loss of messages in the wait queue. This is not a problem in *MOD since every message is quaranteed a corresponding process.

Priority. The message-oriented languages have no way to force an executing task to recognize a high-priority message. If the priority were associated with the task as in *MOD, the message-oriented approach could distinguish among messages to competing tasks but not among different "accept" points within a single task. The reason is that even if the task could recognize the new message, it would be illegal to duplicate the task to accept it. Trying to associate a priority with the message is also futile because there is no way to force the executing task to perform the corresponding "accept" statement.

Simultaneous Messages. Consider the "get" process in Figure 2. If three simultaneous messages arrive, three independent "get" processes would be created which could all be in execution on a multiprocessor system. In the message-oriented approach, each

task can process one message at a time and it is illegal to duplicate a task; therefore, software is automatically biased toward single processor systems. In *MOD, the distinction was intended to be transparent.

Synchronization Primitives. The ADA semaphore example is typical of a synchronization primitive constructed using messages. The disadvantage is that a task must be created for each semaphore. Additional problems arise if the order of message arrival forms the basis for synchronization. Since separate queues are maintained for parallel "accept" statements, the sequence information is lost. In all fairness, we will also agree that synchronization primitives are less useful in a message system since the "accept" statement is an exclusion mechanism.

3.3 Other Languages

Hoare's CSP, DoD's ADA, and Feldman's PLITS language are all message-oriented. CSP and PLITS are oriented toward end-user programming. PLITS, for instance, performs automatic routing of messages which would be a user-implemented service in *MOD. CSP is strongest in its exploration of nondeterministic programming features while PLITS is more completely specified with respect to distributed programming. The ADA language is a CSP derivative which, if used for systems programming, suffers from the defects listed earlier.

*MOD extends Brinch Hansen's "distributed processes" for network communication and improves the multiprogramming features of Wirth's Modula language. For example, each "distributed process" can be encoded as a *MOD "processor module" as follows:

```
process distributed
     <own variables>
     proc name(input params#output params)
        <local variables>
        <statement>
     <initial statement>

processor module *MODdistributed;
     define name, nametype;
     <own variables>
     type nametype= record
                    output params
                    end record;
     process name(input params):nametype;
        <local variables>
        begin <statement>
        end name;
     begin <initial statement>
     end *MODdistributed.
```

The *MOD extensions to these languages are summarized below:

1. Added processor and network module notation.
2. Defined network-wide, strong type checking.
3. Defined a separate compilation facility for network construction.
4. Added "size" specifications to types for interface definitions.
5. Deleted "device modules".
6. Integrated priority into signal and process handling.
7. Changed signal and process semantics to conform to network usage.
8. Improved external interface specification statements.
9. Added data abstraction primitives and parameterized types.

In addition, we have retained a "degree of transparency"[21] between concurrent and distributed programming features while trying to satisfy the efficiency constraints of a systems programming language.

4. System Construction and Testing

*MOD has more severe binding problems than found in most systems in that a network can be separately compiled by processor modules, processor modules by modules, and modules by even smaller modules. All must be type checked and loaded to form a working system. The compilation order rule is that all declarations must be available prior to a module's compilation. A module with the "external" clause is termed a program stub which can be defined by the programmer or created by the *MOD compiler. The former option can be used for top-down program creation or to interface with foreign systems. Once the corresponding module body is defined as in Figure 2, the *MOD compiler automatically gen-

erates a program stub in textual form. There are several advantages to this approach.

First, the programmer only maintains one object--the program; this may be contrasted with other languages which require the user to maintain both a declarations package and a program body. Since the *MOD stub is generated from the most recent copy of the program, the probability of inconsistencies between the two definitions is minimal. Finally, the *MOD method automatically generates a program specification in a machine-independent format which is useful if software is being developed at several different sites.

Once the object modules for a processor module have been created, a program called the binder is invoked which checks the creation dates for consistency and releases the object modules to the system loader to create a bootable core image file. The core image contains the machine-dependent *MOD kernel discussed in the next section and a compiler-created table which contains the message codes for external communication and the process addresses for arriving transmissions. At this point, the software can be tested.

In order to test software in the distributed environment, it must be possible to experiment both with software algorithms and hardware components. The advent of high-level languages has greatly enhanced algorithm development but the same flexibility is not present for hardware. The virtual machine approach [11] was a step in the right direction but was primarily oriented towards the construction of multiprogrammed, single processor software. The VM environment has been suggested [26] as suitable for the development of network software but the user is still

given a bare machine as a starting point.

Our proposal consists of a two-level approach-- 1) a VHLN(virtual high-level language network) *MOD environment for network software development and experimentation on a single host computer (a PDP11 or VAX in the current implementation effort); and 2) a compiler capable of producing bootable code for a number of different machines. The VHLN system is a simulated network which uses compiled code and runtime aids to provide a variety of error-checking and debugging aids. Thus, hardware/software systems can be developed in an economical manner on a single processor, even if the target hardware is not available due to delivery or design problems. The *MOD system can also be used as a research tool by institutions that only have a single processor. Once a software system has been tested, it can be moved to the host machines for production use.

Several advantages ensue from the use of a VHLN environment. First, as is illustrated in the Dining Philosophers example, there are no machine details to muddle up the solution. In fact, if machine designers made peripheral devices maintain the process/signal philosophy, no such bit twiddling would ever be necessary! The high-level language methodology provides system-wide type-checking, data abstraction, and encapsulation mechanisms. In addition, the "defining module" concept insures that someone is responsible for every bit of software in the system; if something does not work, the culprit is easily found. Also, software can be developed in an environment that provides a useful set of diagnostic and debugging tools before being moved to the target computer system. It is much more economical to develop software aids for one host development computer than for

each target machine. The VHLN system also has some disadvantages.

The simulated, multiple processor environment provided by the *MOD system is unrealistic in the sense that events will not have a real-time correspondence to the performance of systems running on bare machines. It might be possible to add some clock-driven primitives to the language to address this problem if it proves serious. As we gain more experience with the system, it will be possible to draw more definitive conclusions regarding the ease of moving from a simulated to a real network.

5. The *MOD Kernel

The *MOD kernel performs process control and message transmission functions only; any routing, security, buffering, or flow control operations are the domain of the systems programmer. In addition, process control does not include the traditional scheduling decisions. A process loses control of a hardware processor only by terminating, blocking, lowering its priority, or by receipt of a message for a higher priority process. Therefore, the major kernel function is message processing which is controlled by a compiler-generated table.

A list is generated by the compiler which details the processes referenced and the corresponding processor identification; in addition, a similar list is created for local processes that occur in "export" or "define" statements. When an external process request occurs, the kernel constructs a message from the argument list as follows:

```
var message: record
        arg1: T1;
```

:

arg_N: T_N;

end record

The processor identification for the destination is available from the kernel tables so the message is transmitted along with some control, error, and sequence information. When the message is received, the target processor verifies that the process has been exported, creates an activation record containing the message for the new process, and sets the "origin" time stamp. If the new process has the highest "ready" priority, it will begin execution immediately. For a functional process call, the originating kernel changes the process status of the caller to indicate that it is waiting for a reply message from the target processor. When a reply arrives, its origin is verified against that in the process' control block; the reply is appended to the process' activation record; and the process' status is changed to "ready". It will resume execution if it has the highest priority.

At this point, it should be noted that most I/O processors do not have the sophistication or the flexibility to implement the previous protocol. Wirth [27] proposed the concept of a "device" module to capture the essence of I/O programming. By necessity, the device module's syntax is very machine and, in some cases, device dependent. For instance, a PDP-11 module contains priority level, interrupt location, and device register syntax. Device modules have been omitted from *MOD because of their inherent machine dependency; instead, process calls and signals are used to their fullest capabilities. The advantage is

that all programs retain their machine independence at the expense of a small number of machine language instructions for each processor which convert device interrupts to signals or process calls, as appropriate. Consider the following example from Wirth [28].

```
device module timing[6];
    define tick;
    use time;
    var tick: signal; LCS[177546b]: bits;
    process driver[100b];
       begin
       LCS[6]:=true;(*start clock running*)
       loop
            doio;  inc(time);
            while awaited(tick) do
                  send(tick);  end;
             end (*loop*);
      end driver;
    begin driver
    end timing;
```

This example would be expressed in *MOD as follows.

```
  process timing;
      import tick,time;
      const STARTCLOCK=4;
      var   interrupt:signal;
      begin
```

```
priority:=6;
   (*start clock*)
sys(STARTCLOCK,interrupt);
loop
   wait(interrupt);  inc(time);
   while awaited(tick) do send(tick);
        end while;
   end loop;
end timing;
```

The "sys" procedure is the only escape mechanism to the host machine. In the example, the argument STARTCLOCK selects a kernel routine which initializes the clock interrupt location(100_8) and starts the clock running. Since the *MOD process is waiting at priority six, the "interrupt" signal will wake it up and the priority will prevent processor preemption by messages to lower priority processes. The priority construct provides the same functionality as processor priority and has the advantage of being dynamic as opposed to the static notation of the device module. Another advantage of the high-level approach is that device handlers can be easily debugged by testing their response to artificially generated signals. Device modules have one advantage in that all the information is available, but force the programmer to accede to the perversity of the hardware designer (see Wirth [29] for a list of typical problems). Since recent hardware trends indicate a growing support of high-level language architecture, there is no reason why this philosophy should not also be extended to external communication interfaces.

This implementation also makes it easy to program hierar-

chies of interrupt handlers as proposed by Lampson [17]. Consider the folllowing example taken from Unix [22].

The Unix system is initialized so that a memory fault trap goes to an error routine. However, to create the table of free space the system needs to know the current memory size. This is accomplished by doing successive stores until a memory fault occurs; the point of the fault identifies the highest available memory address. The problem occurs in redirecting the trap away from the error routine. The *MOD solution is to execute a "wait (memoryfault,2)" which supersedes the "wait (memoryfault,1)" in the error routine. When the signal occurs, the trap is diverted, as desired. The UNIX solution is not nearly so elegant.

We have tried to illustrate by these examples that the distinctions between I/O programming and process communication are largely artificial; therefore, one methodology will suffice for both.

6. Conclusions

The *MOD system represents an exploration of the design decisions necessary to apply the modular programming philosophy of Wirth [30] to the development of distributed software and to propose an environment conducive to the construction and debugging of such systems. This paper would not have been written but for the impetus and inspiration of Brinch Hansen's excellent article[3] on distributed processes.

The current *MOD compiler is a 2800 line C[23] program which runs on a PDP11/45 or VAX UNIX system, is one-pass, generates object code, and compiles at 3000 LPM(VAX) or 1000 LPM(11/45). The compiler is table-driven for parsing, semantic analysis, and code

generation and could easily be modified to generate code for other machines. The present implementation is restricted to a single "processor module" and has been well tested by sixty students each of whom implemented a multiprogramming operating system which executed in VM mode. The network version of *MOD is under development.

REFERENCES

[1] Andrews, G.R., "The Design of a Message Switching System: An Application and Evaluation of Modula", IEEE Trans. on Software Engineering 5, 2(March 1979) 138-147.

[2] Ball, J.E., Williams, G.J., Low, J.R., "Preliminary ZENO Language Description", The University of Rochester, TR41, (Jan. 1979).

[3] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Comm. ACM 21, 11(Nov. 1978), 934-941.

[4] Cook, R.P., "An Introduction to Modular Programming for Pascal Users", The University of Wisconsin-Madison, Technical Report, (Jan. 1979).

[5] Cook, R.P., "The *MOD System Guide", in preparation.

[6] Dahl, O.J. et al., "Simula 67 Common Base Language", Norwegian Computing Center, Oslo(May 1968).

[7] Dijkstra, E.W., "Cooperating Sequential Processes", in Programming Languages (F. Genuys ed.), Academic Press, (1968) 43-112.

[8] Dijkstra, E.W., "Guarded Commands, Non-Determinacy and A Calculus for The Derivation of Programs, Marktobersdorf NATO Conference, (Aug. 1975).

[9] Feldman, J.A., "High Level Programming for Distributed Computing", Comm. ACM 22, 6(June 1979) 353-368.

[10] Geschke C.M., J.H. Morris Jr. and E.H. Satterthwaite, "Early Experience with Mesa", Comm. ACM 20, 8(Aug. 1977) 540-553.

[11] Goldberg, R.P., "Survey of Virtual Machine Research", Computer, 6(June 1974) 34-44.

[12] Havender, J.W., "Avoiding Deadlock in Multitasking Systems", IBM Sys. J. 7, 2(1968) 74-84.

[13] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Comm. ACM 17, 10(Oct. 1974) 549-557.

[14] Hoare, C.A.R., "Communicating Sequential Processes", Comm. ACM 21, 8(Aug. 1978) 666-677.

[15] Honeywell, Inc. and Cii Honeywell Bull, "Reference Manual for the ADA Programming Language", SIGPLAN Notices 14, 6(June 1979) Part A.

[16] Honeywell, Inc. and Cii Honeywell Bull, "Rationale for the Design of the ADA Programming Language", SIGPLAN Notices 14, 6(June 1979) Part B.

[17] Lampson, B.W., "Dynamic Protection Structures", AFIPS FJCC, (1969) 27-38.

[18] Lampson, B.W. et al, "Report on the Programming Language Euclid", SIGPLAN Notices 12, 2(Feb. 1977).

[19] Lauer, H.C. and Needham, R.M., "On the Duality of Operating System Structures", Proc. Second Int. Symp. On Operating Systems Structures, IRIA(Oct. 1978).

[20] Parnas, D.L., "A Technique for Software Module Specification with Examples", Comm. ACM 15, 5(May 1972) 330-336.

[21] Parnas, D.L. and Siewiorek, D.L., Use of the Concept of Transparency in the Design of Hierarchically Structured Systems", Comm. ACM 18, 7(July 1975) 401-408.

[22] Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System", Comm. ACM 17, 7(July 1974) 365-375.

[23] Ritchie, D.M., "C Reference Manual", Bell Labs, (Jan. 1974).

[24] Shaw, M., Wulf, W.A., London, R.L., "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", Comm. ACM 20, 8(Aug. 1977) 553-564.

[25] Watson, R.W., Timesharing System Design Concepts, McGraw-Hill, (1970).

[26] Winett, J.M., "Virtual Machines for Developing Systems Software", Proceedings IEEE Computer Society Conference, Boston MA, (Sept. 1971).

[27] Wirth, N., "Modula: A language for Modular Multiprogramming", Software- Practice and Experience 7, 1(1977) 3-35.

[28] Wirth, N., "The Use of Modula", Software- Practice and Experience 7, 1(1977) 37-65.

[29] Wirth, N., "Design and Implementation of Modula", Software-Practice and Experience 7, 1(1977) 67-84.

[30] Wirth, N., "Toward a Discipline of Real-Time Programming", Comm. ACM 20, 8(Aug. 1977) 577-583.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 2008 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) *MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING		5. TYPE OF REPORT & PERIOD COVERED Summary Report - no specific reporting period 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert P. Cook		8. CONTRACT OR GRANT NUMBER(s) DAAG29-75-C-0024 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Work Unit #8 - Computer Science
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P. O. Box 12211 Research Triangle Park, North Carolina 27709		12. REPORT DATE October 1979 13. NUMBER OF PAGES 27
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

distributed programming, modula, processor module, data abstraction, programming languages

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

Distributed programming is characterized by high communications costs and the absence of shared variables and procedures as synchronization tools. *MOD is a high-level language system which attempts to address these problems by creating an environment conducive to efficient and reliable network software construction. The concept of a processor module is introduced as well as a methodology for distributed data abstraction and process communication. In addition, a VHLN (virtual high-level language network) is proposed for system development.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)