UNIVERSITY OF WISCONSIN
COMPUTER SCIENCES DEPT.
1210 WEST DAYTON STREET
MADISON, WI 53706

*MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING

by

Robert P. Cook

Computer Sciences Technical Report #385

April 1980

*MOD--A LANGUAGE FOR DISTRIBUTED PROGRAMMING

ABSTRACT

Distributed programming is characterized by high communications costs and the inability to use shared variables and procedures for interprocessor synchronization and communication. *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. Several of the *MOD distributed programming constructs are discussed as well as an interprocessor communication methodology. Examples illustrating these concepts are drawn from the areas of network communication and distributed process synchronization.

Key Words and Phrases: distributed programming, Modula, processor module, programming languages, computer networks.

The author is partially supported by U. S. Army Contract DAAG29-75-C-ØØ24 and National Science Foundation grant MCS-79Ø3947.

1. Introduction

*MOD(starmod), a language derived from Modula[35,36,37], is intended for systems or application programming in a network environment. The *MOD design is based on experience with our PDP-11/VAX Modula compiler[7] and was inspired by Brinch Hansen's "distributed processes" concepts[3]. A distributed program can be characterized as an algorithm which requires multiple processors for its implementation; whereas a concurrent program requires multiple processes for its implementation. A sequential program is an algorithm which is implemented using only procedural constructs.

It is interesting to observe that after 25 years of experience with sequential programming, there is still no consensus on language features. The current discussions involving Ada[19,20] provide a case in point. Since even less is known about the requirements for distributed and concurrent programs, *MOD was developed as an experimentation media and is not presented here as the ultimate solution. The primary design criteria were transparency, extensibility, and program adaptability.

Following Parnas' definition[29], a language is transparent with respect to a given computer system if any system state and any sequence of system states which could be obtained by programming the component machines could also be obtained by using the high-level language. For instance, experimentation would be severely hampered if the language were inflexible with respect to process definition and scheduling. Therefore, *MOD provides low-level operations which can be extended by the programmer to define high-level abstractions appropriate to a particular

machine environment. *MOD programs can also adapt to changes in workload or response time requirements. The language is designed so that a library module could be upgraded from a collection of procedures to a collection of processes without changing any of its user's programs. Finally, *MOD permits the user to organize his/her program to reflect the structure of the physical processors and the connectivity of the network.

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[19], Hoare's Communicating Sequential Processes(CSP)[18], Feldman's PLITS[1,13], and Brinch Hansen's "distributed processes"[3]. In particular, we address the distributed programming problem areas of interprocessor communication, software testing and kernel efficiency. The *MOD design decisions in other areas such as data abstraction[8] and synchronization[9] are described elsewhere.

2. Program Organization

Before proceeding further with a more detailed discussion of the *MOD distributed programming mechanisms, the module concept of Modula[35] will be considered as a focal point for network software development. A module usually corresponds to a program abstraction and consists of an external interface specification, data structure definitions, procedures, processes, and an optional initialization part. In *MOD, a module can be used as a type definition or to delineate a lexical scope as in Modula; therefore, both the information-hiding properties proposed by Parnas[28] and the flexibility of the Simula[10] "class" mechanism are maintained.

In Modula, if the prefix "device" or "interface" is used before the "module" keyword, the semantics of the module change. For instance, "interface" denotes a module which semantically is similar to a monitor[17]. *MOD extends the Modula prefix notation to provide the user with the following module types.

```
PROGRAM ::= network module IDENTIFIER [= LINKLIST];
                [MODULEBODY]
                end IDENTIFIER.

PROCESSORDECLARATION ::= processor module IDENTIFIER
                              [REPLICATIONCOUNT];
                         [MODULEBODY]
                         [begin STATEMENTLIST]
                         end IDENTIFIER

MODULETYPEDECLARATION ::= type IDENTIFIER = [MODULETYPE] module;
                              [MODULEBODY]
                              [begin STATEMENTLIST]
                              end IDENTIFIER

MODULEDECLARATION ::= [MODULETYPE] module IDENTIFIER;
                      [[MODULEBODY]
                       [begin STATEMENTLIST]
                       end IDENTIFIER]

MODULETYPE ::= MODULETYPEID | access | scheduler

LINKLIST ::= LINK [,LINK]...
LINK ::= (PROCESSORVAR [,PROCESSORVAR]...)

REPLICATIONCOUNT ::= '[' EXPRESSION ']'
```

A network module is required to define processor connectivity and to declare any types, constants, procedures or processes which are global to the processor modules. The LINKLIST details the communcation paths among processor modules in terms of a source specification and a list of destinations. Variable declarations and the initialization STATEMENTLIST are prohibited at the network level since there is no instruction or data storage outside of a physical processor. Procedures and processes are allowed within network modules for global standardization but are copied automatically by the compiler into each processor module. Figure 1 gives an example of a five processor module "star" system.

Processor modules were introduced as part of the *MOD design to allow the programmer to partition a computation into collections of processes. All procedures and processes within a processor module can directly access shared variables; whereas interprocessor references must be in the form of messages. By using combinations of processes and the ability to specify processor modules, a *MOD user can take full advantage of the physical hardware available in order to exploit the inherent parallelism in his/her algorithm. In some situations, an algorithm may contain more processor modules than the number of physical processors available. In this case, a mechanism such as Jones' "task force" language[21] must be used to map the *MOD virtual processor set onto the physical network. It may also be advantageous to use a mapping specification for networks with non-uniform communication costs. In addition, the performance of processes within a processor module can be improved by mapping the module

Figure 1

```
network module star = (center,pr),(pr,center);
    const NOPROCESSORS = 4;
    processor module center;
        define communicator;
        process communicator(...);
                    .
                    .
                    .
          pr[i].communicator(...); (*call processor i*)
                    .
                    .
                    .
         end communicator;
       begin (*initialization*)
       end center;
    processor module pr[NOPROCESSORS]; (*four peripheral processors*)
        define communicator;
        import center;
        process communicator(...);
                  .
                  .
                  .
          center.communicator(...); (*call center*)
                  .
                  .
                  .
          end communicator;
        begin (*initialization*)
        end pr
   end star.
```

to a multiprocessor. 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.

The module "type" declaration can be used to construct extended data types as in Simula[1Ø] except that a *MOD programmer has more control over external interface and protection specifications. The Modula prefix notation for modules has been extended to include user defined module types in addition to the "access" and "scheduler" keywords. An "access"[8] prefix can be used to define array abstractions while the "scheduler"[9] prefix can be used to build synchronization abstractions such as monitors[17] or interface modules[35]. When a module "type" is used as a prefix to a module declaration, the body of the "type" is automatically replicated in the new module. Next, the syntax of a MODULEBODY will be illustrated.

```
MODULEBODY ::= EXTERNALINTERFACE
               BLOCKHEADING

EXTERNALINTERFACE ::= [define ELEMENT[,ELEMENT]...;]
                      [export ELEMENT[,ELEMENT]...;]
                      [pervasive ELEMENT[,ELEMENT]...;]

BLOCKHEADING ::= [import IDENTIFIER[,IDENTIFIER]...;]
                 [DECLARATIONLIST]

ELEMENT ::= IDENTIFIER[(readonly|protected)]
```

A module boundary delineates a closed lexical scope which can only be superseded by the explicit specification of "define",

"export", "import" or "pervasive". The external interface for a processor module usually lists any message types and process names which are used for communication.

An IDENTIFIER specified in an "import" list causes a declaration from a global lexical 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 the enclosing and all nested levels where the same name is not already declared. The latter option is most useful for making functions appear to their users as "builtin" to the language. For instance, the "sine" and "cosine" functions are declared as "pervasive" in the math library module; therefore, any use of that library automatically imports the math routines into the enclosed scopes.

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 in the Simula[10] "class" notation. "define" is used to reduce the size of "import" lists since "import"ing a module name permits references to any of its "define"d symbols. Secondly, a qualified reference to a "define"d name serves a useful documentation function. 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[14], Euclid[23], Alphard[32], Ada[19], etc.

The *MOD user can also restrict variables and types to read-only access or to no access. These restrictions are not applied within the exporting module. Since the access checking

is enforced by the compiler, it can be maintained even across distributed processors. The only exception is that any restricted variable or instance of a restricted type can be passed as an argument to a procedure exported from the defining module.

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. We have adopted the traditional[33] definitions that 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 state vector which controls and defines the virtual processor on which the process runs. It should be noted that a procedure cannot execute except as part of a process. Since *MOD programs are intended to run on a bare machine, a *MOD process is a much simpler entity than the processes found in most operating systems. For instance, the state vector for a UNIX[30] process on a PDP-11 contains over 1000 bytes of information while the *MOD kernel for the same computer uses 32 bytes for a process' state vector. Processes in the same processor module can communicate using shared variables or messages; however, interprocessor communication can only consist of messages.

Messages are assumed to range from no content(signal or interrupt) to arbitrary data structures. We will refer to the recipient of a message as the message handler, or handler. Furthermore, *MOD enforces strong type checking on messages both within and across processors to maintain system consistency. Finally, any communication mechanisms presented should be

efficient and should not constrain the options of the programmer. In the next sections, the *MOD design will be discussed along with a detailed analysis of the alternatives, advantages and disadvantages.

3.1 Message Handlers

A procedure is a simple example of an intraprocessor message handler. Since a message can be an arbitrary data structure composed of many subfields, the syntax for the argument list and returned value was expanded to permit record and array types. Thus, the familiar argument list notation is used to represent the component fields of a message.

```
PROCEDUREDECLARATION ::= procedure IDENTIFIER[(FORMALS)]
                                        [:TYPEID];
                                    BLOCKHEADING
                                    [begin STATEMENTLIST]
                                    end IDENTIFIER
PROCEDUREREFERENCE ::= PROCEDUREID[(ARGUMENTS)]
```

A procedure is an example of a dependent service in that the procedure's activation record must be attached to the activation record stack of an existing process, usually the caller, in order to execute. *MOD does not permit procedures as interprocessor message handlers. As an alternative, *MOD uses a process-oriented communication methodology.

3.1.1 Processes

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

```
PROCESSDECLARATION ::= process IDENTIFIER[(FORMALS)]
                           [ '[' PRIORITY ']' ]:TYPEID;
                       BLOCKHEADING
                       [begin STATEMENTLIST]
                       end IDENTIFIER
PROCESSREFERENCE ::= PROCESSID[(ARGUMENTS)]
```

Except for the keyword "process" and the optional PRIORITY expression, the declaration is identical to that of a procedure. This correspondence was made intentionally to make it easier for a programmer to convert a sequential program to a concurrent or distributed program. A PROCESSREFERENCE must specify a list of arguments corresponding exactly in type and number to the FORMALS. Every procedure call creates a new activation record; however, a PROCESSREFERENCE creates a new activation record stack and state vector. Thus, each process can have multiple activations all executing in parallel.

The returned value for a functional process is set by assignment to the process identifier and must match the specified TYPEID. A reference to a functional process within the same processor is the same as a procedure reference. However, an interprocessor functional reference implies independent execution for the called process; the caller must wait. Also, the compiler will flag a process reference as an error if there are no direct communication links between the caller and handler processes.

The optional PRIORITY must evaluate to a compile-time constant which specifies the initial(default zero) priority of the process. The initial priority can be modified by altering the value of the pseudo-variable "priority". Priority can be used to

improve the performance or response time of an algorithm by establishing precedence relationships among its processes.

Consider the following execution options for a process as a message handler.

1.	execute in parallel with caller	no reply message
2.	execute in parallel with caller	reply message
3.	caller waits for completion	no reply message
4.	caller waits for completion	reply message

Option 1 is implemented by a *MOD process which does not return a value while Option 4 is represented by a functional process. The reply message is the returned value which can range from a simple variable to an arbitrary record. Option 3 is easily programmed by using Option 4 to return a completion indication. Option 2, which corresponds to Conway's[5] "fork" and "join" methodology, is available using the "port" mechanism described in Section 3.1.2.

The example in Figure 2 illustrates these concepts with a ring network version of Dijkstra's Dining Philosophers[11] problem. As in the original version, five philosophers are each trying to eat from a plate of special spaghetti which has been placed in the middle of a round table. The spaghetti is special since each philosopher requires two forks to eat it. In our example, each philosopher can directly control only 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, messages must be sent around the ring(table) to get permission to use the

Figure 2

```
network module diningroom=(phil[1],phil[2]),(phil[2],phil[3]),
                          (phil[3],phil[4]),(phil[4],phil[5]),
                          (phil[5],phil[1]); (*ring network*)
   processor module phil[5];
        define get, put, got; (*communication processes*)
        module at_table;
            export get, put, got, getforks, putforks;
            import phil;
            var myfork, gotone: semaphore;
            process get(who, fork: integer); (*get "fork" for "who"*)
                begin
                if fork <> pid then phil[pid mod 5 + 1].get(who,fork)
                               else p(myfork);   got(who)
                               end if
                end get;
            process put(fork: integer);  (*give "fork" back*)
                begin
                if fork <> pid then phil[pid mod 5 + 1].put(fork)
                               else v(myfork)
                               end if
                end put;
            process got(who: integer) [1];  (*let "who" use fork*)
                begin
                if who <> pid then phil[pid mod 5 + 1].got(who)
                              else v(gotone)
                              end if
                end got;
            procedure getforks;  (*philosopher waits for both forks*)
                begin   get(if pid=1 then 1 else pid-1);  p(gotone);
                        get(if pid=1 then 5 else pid);    p(gotone)
                end getforks;
            procedure putforks;  (*give forks back and don't wait*)
                begin  put(pid);
                       put(if pid=1 then 5 else pid-1)
                end putforks
            end at_table;
        begin  loop
                (*think*)  getforks;
                (*eat  *)  putforks
               end loop
        end phil
   end diningroom.
```

left-hand fork and to give it back. The algorithm is based on an ordered resource allocation strategy developed by Havender[16] which prevents deadlock and starvation.

Each philosopher is required to call "getforks" to start eating and "putforks" to stop. The lower level protocol as well as the problem restrictions and network topology are completely hidden from the user. The "get" process accepts fork requests and either passes the request to the right in the ring or else gets control of its own fork and sends an acknowledgment to the "who" philosopher. "pid" is a builtin function which identifies each processor by its index value. "got" acknowledges fork allocations and "put" frees forks. The algorithm works correctly without process priority but we have specified a higher priority for the "got" process to illustrate how performance can be "tuned" by such assignments. Note that every message in the ring activates a parallel process; thus, the performance of the algorithm can also be improved by using multiprocessors for the nodes of the ring.

3.1.2 Ports

In addition to the call/return form of message communication described in Section 3.1.1, it is often desirable to implement coroutine[6], rendezvous[19], or multiple handler protocols. We have integrated Balzer's[2] port mechanism into the *MOD design to provide these facilities.

```
PORTDECLARATION ::= port IDENTIFIER[(FORMALS)][:TYPEID];
PORTREFERENCE ::= PORTID[(ARGUMENTS)]
```

A procedure provides a dependent message handler; a process is an

independent message handler; and a port is a message queue which is independent of a handler. Thus, a port can be thought of as a queue of procedure or process messages(argument lists) waiting for execution. With a procedure or process, the caller initiates execution; with a port, only the handler can initiate execution as follows:

```
REGIONSTATEMENT ::= region PORTID [,PORTID]... ;
                        [PORTBODY]
                    end region
PORTBODY ::= STATEMENTLIST | PORTCASE [;PORTCASE]...
PORTCASE ::= PORTID: begin STATEMENTLIST
                    end PORTID
```

If only a single PORTID is present, the PORTBODY must be a STATEMENTLIST; otherwise, a PORTCASE is required for each PORTID listed. The latter option permits a process to service requests with non-deterministic arrival times. The region statement copies the selected message into the current activation record and reserves a return value cell if the port is a function. When an "exit" from the region occurs, the returned value, if present, is transmitted to the PORTREFERENCE statement. As with processes, the caller only waits if the port returns a value. At compile-time, the scope of the argument list is opened on region entry and closed on region exit for a single port. For multiple ports, a scope is opened and closed for each PORTCASE. When the region statement is executed, the executing process will wait until a message is present at one of the listed ports. If multiple regions are waiting for the same port when a message arrives, the

region to execute is chosen non-deterministically. If a region is waiting for several ports which simultaneously receive messages, the arrival order determines the processing order. Also, an "awaited" function is provided to test the status of a port.

Figure 3 illustrates an implementation of the previous example using ports. The first example uses a process for every message which provides the maximum parallelism. In the port implementation, only one communication process is created for each processor and the messages are handled sequentially. However, multiple "communicator" processes could still be invoked to increase parallelism.

3.2 Processor Communication

In *MOD, processes, ports and procedures are all referenced with the same syntax; thus, the user of a module's services need never know how the service is provided. The advantage is that the implementation can be easily modified to adapt to changes in workload or response time requirements. The only restriction on such changes is that procedures cannot be used for interprocessor communication.

A procedure invocation implies the creation of an activation record which is added to the stack of the calling process. However, for an interprocessor call, the stack of the calling process is not available. The alternative is to attach the procedure's activation record to the stack of a randomly selected process in the called processor. We rejected this choice because the selected process could not continue execution until the procedure's activation record was removed. A more serious problem occurs if the process controls a resource needed by the

Figure 3

```
network module diningroom=(phil[1],phil[2]),(phil[2],phil[3]),
                          (phil[3],phil[4]),(phil[4],phil[5]),
                          (phil[5],phil[1]);  (*ring network*)
   processor module phil[5];
        define get, put, got;
        module at_table;
            export get, put, got, getforks, putforks;
            import phil;
            var myfork, gotone: semaphore;
            port get(who, fork: integer);  (*get "fork" for "who"*)
            port got(who: integer);        (*let "who" know*)
            port put(fork: integer);       (*give "fork" back*)
            process waitrep(who: integer); (*local rep. for "who"*)
                begin  p(myfork);  got(who)
                end waitrep;
            process communicator;
                begin loop
                      region get, put, got;
        get:            begin
                        if fork<>pid then phil[pid mod 5+1].get(who,fork)
                                    else waitrep(who)
                                    end if
                        end get;
        put:            begin
                        if fork<>pid then phil[pid mod 5+1].put(fork)
                                    else v(myfork)
                                    end if
                        end put;
        got:            begin
                        if who<>pid then phil[pid mod 5+1].got(who)
                                   else v(gotone)
                                   end if
                        end got
                      end region
                end loop
            end communicator;
            procedure getforks;  (*philosopher waits for both forks*)
                begin   get(if pid=1 then 1 else pid-1);  p(gotone);
                        get(if pid=1 then 5 else pid);    p(gotone)
                end getforks;
            procedure putforks;  (*give forks back and don't wait*)
                begin  put(pid);
                       put(if pid=1 then 5 else pid-1)
                end putforks;
            begin   communicator
            end at_table;
        begin loop
              (*think*)  getforks;  (*eat *) putforks
              end loop
        end phil
    end diningroom.
```

procedure; in this case, deadlock could result. Therefore, *MOD provides only ports and processes for interprocessor communication.

A process provides parallel execution for each message handler but requires a storage allocation operation to create its stack. A port, or multiple ports, can be handled by a single process which remains permanently active; thus, a port communication involves only a context switch which can be performed very quickly with modern hardware. If the handler execution time is considerably longer than process creation time, a process is the appropriate implementation choice. Otherwise, a port will provide the fastest response time.

3.3 Comparison with Other Languages

*MOD integrates Brinch Hansen's[3] "distributed processes"(DP) concepts with the modular programming philosophy of Wirth[35,36,37]. Figure 4 illustrates the compatability of a DP "process" with a *MOD "processor module". *MOD also provides the user with network and processor modules, ports, processes, and synchronization abstractions[9]. DP is oriented towards a single process per processor while *MOD is intended as a general-purpose language for distributed programming.

3.3.1 Ada

Ada[19,20] is a language proposed by the Department of Defense for universal use; therefore, it should be instructive to compare an Ada "task" with a *MOD "process". In *MOD as in Modula, a process can not be declared inside other processes or procedures. An Ada task "may be declared local to other task

Figure 4

```
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;
```

bodies, packages, subprograms and blocks" and a "thread of control" is maintained between a task and its parent. In addition, "any subprogram, module, or block containing task declarations cannot be left until all local tasks have terminated". Furthermore, tasks can contain procedures which can be called by other tasks. Thus, tasks within blocks or procedures must be able to share activation record stacks and tasks which call procedures in other tasks must be able to share the procedure's global variable space. *MOD avoids these addressing and protection problems by allowing shared access only to global variables. *MOD also avoids maintenance of the "thread of control" information. All of these Ada conventions are even more difficult to implement across processors.

It is an error to initiate more than one instance of an Ada task; the user can declare vectors of tasks but each element must be individually addressed. Ada provides ports in the form of "entry" declarations which are restricted to a task body. Therefore to communicate with a task, the user must name an entry point. If multiple instances of tasks with the same name were allowed, the entry point could not be named without ambiguity. *MOD avoids this restriction by requiring port declarations to occur outside of processes; thus, any port can be serviced by any process. This is not possible in Ada.

The Ada "accept" statement is equivalent to a *MOD "region" but always requires the caller to wait until the body of the "accept" completes execution. Thus, Ada does not have a direct equivalent to *MOD functional ports, functional processes, or multiple processes. Procedures, processes, and ports are refer-

enced identically in *MOD; only procedure and entry references are similar in Ada.

3.3.2 Mesa

Mesa[27] includes both processes and ports in a format similar to *MOD. Mesa allows a procedure body to be called as a process by using the "fork/join" statements. The "join" statement allows the returned value to be retrieved after the call. We decided not to use this methodology because it would require changes to all user programs if a handler were changed from a procedure to a process. As pointed out in a recent paper[22], the "join" construct is infrequently used compared to the "detached" process option which is similar to the *MOD design. Ports in Mesa were designed to implement coroutines; procedures must have a special "start" phase, be "connect"ed to a port and then "restart"ed. The Mesa port mechanism is so complex because it was designed to set up a coroutine relationship between two or more procedures within the same process. The *MOD port is a more general mechanism which has a variety of uses including connecting processes as coroutines.

3.3.3 CSP and PLITS

Hoare's CSP[18] and Feldman's[1,13] PLITS languages are oriented toward end-user programming on top of an operating system. PLITS, for instance, performs automatic routing and flow control of messages; these would be user-implemented services in *MOD. CSP is strongest in its exploration of nondeterministic programming features while PLITS is more completely specified with respect to distributed programming. Both langauges are port

oriented. Hoare's processes can be created dynamically as in *MOD; however, the relationship between two communicating processes is symmetrical and requires both of them to name the other. Communication is asymmetrical in *MOD so that in hierarchical systems, handlers can be created without knowing the identities of their callers.

3.3.4 CLU

Liskov[24] has recently proposed some distributed computing extensions to CLU[25]. A CLU "guardian" is similar to a *MOD "processor module" but is restricted to ports for communication. There is no equivalent to a "network module" for processor organization. Ports are global to processes as in *MOD and include timeout and error detection notation. The major difference between the two proposals arises because of the integrated definitions of procedures, processes, and ports in *MOD. All of the languages that we have examined, including CLU, have a special syntax for one or more of these constructs; thus, any change to a library program may require changes to all user programs.

4. Software Testing

In order to test software in the distributed environment, it must be possible to experiment both with software algorithms and hardware organizations. The advent of high-level languages has greatly enhanced algorithm development but the same flexibility is not present for hardware. The virtual machine approach[15] 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[34] 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 testing on a single host computer (a PDP-11 or VAX in the current implementation effort); and 2) a compiler capable of producing code for a number of different machines. The VHLN system is a runtime package which executes all of the processor modules on a single processor. The package also provides debugging, simulation and performance analysis aids. It is much more economical to develop software tools for one host development computer than for each target machine. Once a software system has been tested, it can be moved to the network for production use. We have used this methodology to construct operating systems for virtual processors, file systems, and simulations of the Ethernet[26] and DCS[12] networks.

The VHLN system also has some disadvantages. The simulated, multiple processor environment provided by the *MOD test system is unrealistic in the sense that events will not have a real-time correspondence to the performance of systems running on bare machines. However, even this problem can be solved by using the techniques proposed by Canon[4]. As we gain 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 code generated by the *MOD compiler is independent of its execution environment. For example, we have several bare machine kernels for different PDP-11s, a kernel for simulation,

and a kernel which allows *MOD programs to run on UNIX[30]. The bare machine kernels are used for real-time testing, the simulation kernel for performance evaluation, and the UNIX kernel for system development and debugging. Except for simulation statements, the same *MOD program could be executed with any of the kernels. Next, the *MOD bare machine kernel will be described.

The *MOD kernel performs message transmission and synchronization functions only; any routing, scheduling, or flow control operations are the domain of the systems programmer. 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.

The compiler generates a list which details the processes and ports referenced by a processor module together with their processor addresses. In addition, a similar list is created for local ports or processes that have been "export"ed or "define"d. When an external process or port call occurs, the kernel constructs a message from the argument list as follows:

```
var message : record
                arg1 : TYPE1;
                  :
                argN : TYPEN;
              end record
```

The processor address for the destination is available from the kernel tables to control message transmission. For a functional call, the originating kernel changes the process' status to indicate that it is waiting for a reply message from the target pro-

cessor. When a reply arrives, its origin is verified against the saved value 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 when it is the highest priority, "ready" process.

In *MOD, each interrupt or message is sent to either a port or a process. A message to a process creates a new process activation record stack. The new process' priority and start address are taken from the kernel control table. A process switch occurs only if the new process has a higher priority than any of the executing processes. A message to a port is logged if no process is waiting; otherwise, a process switch decision based on priority is made. Port communication only requires a process switch while a process call requires both the creation of an activation record and a process switch. The critical question is how quickly these operations can be performed. Table 1 lists the number of instructions executed for process/port communication on our PDP-11 standalone and UNIX kernels.

The UNIX kernel copies each process' stack segment from high core to a save area on every process switch. The standalone kernel has less work to do on context switches because there is no copying, although the state vector is more complicated. Also, the initial stack space for new processes is fixed in size which means that the allocate/free operations can be executed in constant time.

6. Summary

The *MOD system represents an exploration of the design decisions necessary to apply the modular programming philosophy

Table 1

Operation	Standalone	Unix Kernel
process create(N arguments)	39+7N+A+S	31+2N+A+S
process delete	71+B+S	10+5E+B+S
port call(no arguments)	18+S	14+S
port call(N arguments)	24+7N+S	20+B+A+2N+S
(A) allocate memory	45	20+10C
(B) free memory	5	22+6D
(S) process switch	32	36+B+A+2F+2G

(C) number of free list cells of smaller size

(D) number of free list cells at a smaller address

(E) number of processes in scheduling list

(F) old process' stack size

(G) new process' stack size

of Wirth to the development of distributed software and to propose an environment conducive to the construction and testing of such systems. This paper would not have been written but for the impetus and inspiration of Brinch Hansen's article[3] on distributed processes.

The current design was developed by experimenting with extensions to our Modula compiler which is written in C[31] and runs on Version 7 UNIX. The compiler generates either PDP-11 or VAX11/780 machine code. A separate *MOD compiler is currently under development.

REFERENCES

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

[2] Balzer, R.M., "PORTS--A Method for Dynamic Interprogram Communication and Job Control", Proceedings of AFIPS SJCC Computer Conference 39, (1971) 485-489.

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

[4] Canon, M.D. et al., "A Virtual Machine Emulator for Performance Evaluation", Comm. ACM 23, 2(Feb. 1980) 71-80.

[5] Conway, M.E., "A Multiprocessor System Design", Proceedings AFIPS Fall Joint Computer Conference 24, (1963) 139-146.

[6] Conway, M.E., "Design of a Separable Transition-Diagram Compiler", Comm. ACM 6, 7(July 1963) 396-408.

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

[8] Cook, R.P., "Data Abstraction in *MOD", in preparation.

[9] Cook, R.P., Schedulers as Abstractions--An Operating System Structuring Concept", The University of Wisconsin-Madison, Technical Report, (Jan. 1980).

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

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

[12] Farber, D. et al., "The Distributed Computing System", Proceedings COMPCON 73, IEEE Computer Society, (Feb. 1973) 31-34.

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

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

[15] Goldberg, R.P., "Survey of Virtual Machine Research",

Computer, 6(June 1974) 34-44.

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

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

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

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

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

[21] Jones, A. and K. Schwans, "TASK Forces: Distributed Software for Solving Problems of Substantial Size", Fourth Intl. Conference on Software Engineering, SIGSOFT-ACM, (Sept. 1979) 315-330.

[22] Lampson, B.W. and D.D. Redell, "Experience with Processes and Monitors in Mesa", Comm. ACM 23, 2(Feb. 1980) 105-117.

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

[24] Liskov, B. et al., "CLU Reference Manual", Computation Structures Group Memo 161, Laboratory for Computer Science, M.I.T. Cambridge, Mass., (July 1978).

[25] Liskov, B., "Primitives for Distributed Computing", Proceedings Seventh Symp. on Operating Systems Principles, Pacific Grove, Calif., (Dec. 1979) 33-42.

[26] Metcalfe, R. and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. ACM 19, 7(July 1976) 395-404.

[27] Mitchell, J.G., Maybury, W. and R. Sweet, "Mesa Language Manual", Xerox PARC Technical Report CSL-79-3, (April 1979).

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

[29] 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.

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

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

[32] 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.

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

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

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

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

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