ON THE DESIGN AND SPECIFICATION OF MESSAGE ORIENTED PROGRAMS[+]

by

Paulo R.F. Cunha[*]
Carlos J. Lucena[†]
T.S.E. Maibaum[*]

Research Report CS-79-25[§]

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

[*] Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

[†] Departamento de Informatica
Pontificia Universidade Catolica
Rua Marques de S. Vincente, 225
Gavea-CEP 22453
Rio de Janeiro, RJ, Brazil

[§] Revised, March 1980.

[+] This work was supported by a research grant (to C.J. Lucena) from the J.S. Guggenheim Memorial Foundation and also by the National Sciences and Engineering Research Council of Canada.

A B S T R A C T

Processes are building blocks for the modelling of environments in which parallel and distributed processing occurs. They play in parallel programming the role of standard units (as do subroutines or procedures in sequential programming). Process communication and synchronization can be achieved either through shared variables (common address space) or by message transmission. It has been shown that the message transmission mechanism leads to a more general computational structure. We develop in this paper the beginnings of a methodology to deal with what we call message oriented programming. We note in passing that the methodology for programming with shared variables is well developed and shows a development leading from operational (automata oriented) constructs (semaphores) to high level programming constructs (critical regions and then monitors). Recent mathematical theories of message oriented programming deal with the subject from an operational (automata oriented) point of view. However, the

models are too far removed from the control and data structures of programs to guide the designer in constructing a process. To be able to bridge this gap between program specification and program implementation (expressed in the high level language that we use), we resort to a definitional specification technique based on the concept of abstract data type.

The specification technique is used in conjunction with some useful design principles to illustrate our ideas via solutions to well known problems. The two main principles we discuss are: differences (and needs) for definitions of process and message structures in a given application and the concepts of module strength and module coupling as put forward by Myers. Finally, we illustrate how the high level definitional technique leads us to a straight-forward method for studying some properties of message oriented programs. We give an example of proving the deadlock freeness of a solution to the consumer and producer problem.

Key Words and Phrases: Distributed processing, message oriented programming, software design, specification, synchronization constructs, definitional specification, communication descriptions, deadlock.

1. Introduction

The classical approach for dealing with complex problems and computer systems in particular is to attempt their decomposition into smaller and simpler parts. Processes are building blocks for the modelling of dynamic environments in which parallel and distributed processing occurs. They play in parallel programming the role of standard units which have been reserved for subroutines or procedures in sequential programming.

Although there is hardly any agreement in the literature on a precise definition for the term process, several programming constructs have been proposed to capture its intuitive meaning. The independent actions that occur in a parallel system can be represented through independent processes. The processes are independent in the sense that each contains all the information required for the execution of its intended action. Process communication and synchronization can be achieved either through shared variables or by message transmission. It has been shown [38] that the message transmission mechanism leads to a more general computational structure, since shared variables can be viewed as a special case of message transmission in which two processes cannot communicate at the same time. Furthermore, shared variables cannot deal with the case in which processes run on different nodes of a network of processors because they require a common address space.

In the last few years the various approaches to parallel

programming, from Dijkstra's semaphores [14] to Hoare's monitors [21], responded to the need of good engineering techniques for parallel programs. (In fact they encompassed ideas from structured programming, program verification and programming with abstract data types.) Nevertheless, since all these techniques relied upon shared variables they failed to match up with the present needs for fully distributed systems.

The current trend in parallel programming is programming through messages and processes. The general idea of message passing for interprocess communication was preliminarily discussed by Brinch Hansen in [2]. More recently the concept has been discussed in a more general setting, by presenting processes and messages as both a structuring tool and as a synchronization mechanism. Instances of this recent effort can be found in Zave [38,39], Jammel [23], Hoare[22] and in the description of multiprocessing systems such as Demos [1], Mininet [29] and Thoth [8].

Zave [38] has argued for the naturalness, usefulness and generality of programming with messages and processes. We think that a further characterization of this programming technique is necessary. It needs to be at least as well understood as the techniques for parallel programming with shared variables. In other words, design principles, specification and proof methods need to be developed for the complete characterization of this novel programming style.

In the present paper we review the programming style induced by the use of semaphores [14,37], critical regions [19,3,4] and monitors [5,21] . We focus on the basic ideas introduced by the different methods for the utilization of shared variables for process communication. We then contrast the above approaches to programming with processes and messages which we call message oriented programming. The discussion is based on different solutions to the often used example of the readers and writers, and a message oriented program for the producers and consumers problem. We emphasize the design principles which guide message oriented programming.

The paper proceeds by showing that the needed theoretical foundations for the new programming approach are currently being developed. We illustrate this last point by presenting a concise description of the specification method proposed by Milne and Milne [31] that introduces the notion of flow algebras for the specification of processes and messages. We finally report our current efforts towards an algebraic specification of processes and messages. All the advantages usually attributed to the algebraic specification methods [17,40,16] are present in this context and the method described is proving to be useful in the specification and verification of properties of parallel programs [11].

2. Process Synchronization Through Shared Variables

In the present section we review the notions of semaphores [14] , conditioned critical regions [19,3,4] and monitors [5,21] which have been used for the syncronization of processes through shared variables. The following discussion will be based on different versions of the problem of the readers and writers, originally proposed and solved by Courtois et. al. in [10]. The problem can be stated in the following way. Readers and writers are processes which share a resource. The readers can use the resource simultaneously but the writers require exclusive access to it. When a writer is ready to use the resource, it is entitled to do so as soon as possible.

The discussions which follow the brief definition of each syncronization method and the presentation of the example will center on programming style. The comments made about the various methods are not necessarily new. In fact, each newer approach tried to by-pass the problems recognized in the preceding method. The reason for the compilation of the characteristics of the various methods in the present text is to facilitate the task of establishing a clear differentiation between these methods and the one presented in the next section.

2.1 Semaphores

The concept of semaphore was introduced by Dijkstra [14] and it is known as the first attempt for dealing systematically with inter-process communication in programming.

A semaphore s is a non-negative integer which can only be modified by two special operations called P and V . In his original work Dijkstra differentiated between binary (assuming the values 0 and 1) and general semaphores. The P and V operations are defined in the following way:

$$P(s) \triangleq \text{when } s > 0 \text{ do } s \leftarrow s - 1$$

$$V(s) \triangleq s \leftarrow s + 1$$

In Appendix I we present the solution to the readers and writers problem proposed by Courtois et. al. in [10]. Although other solutions to the same problem also based on semaphores have been published (e.g. Brinch Hansen [3], Keller [24]), Courtois' version of the problem does not resort to extensions of the original semaphore concept.

The following comments can be made about the utilization of the semaphore concept in its original form:

(i) With the semaphore approach, the code for synchronizing each access to a resource is located in the process that requires it. As a consequence the synchronization code gets distributed through all the processes therefore

causing repetition of commands.

(ii) The verification of a control structure based on semaphores is not easy (see for instance Brinch Hansen [3]). The problem resides in the fact that semaphores establish a strong dependency among the various processes. Besides, a system based on semaphores is more susceptible to catastrophic errors since the omission of P or V operation in one of the processes can lead the whole sytem to block.

An important design assumption was made in connection with semaphores: they require all processes to be able to address directly the shared structure.

2.2 Conditional Critical Regions

The concept of conditional critical regions was first introduced by Hoare in [19] and then further studied by Brinch Hansen in [3] and [4]. The proposal was motivated by the contemporary work in structured programming and program verification and was aimed at providing features for structured multi-programming.

For an easy understanding of the example in Appendix II, we quote a few definitions from [4].

(i) var v : shared T ; shared variable v of type T which can only be referrenced and changed inside a critical region.

(ii) region v do S ; associates a statement S with shared variable v . Critical regions referring to the same variable exclude each other in time.

(iii) await B ; delays a process until the components of v satisfy the condition B and must be enclosed in a critical region.

The general form of a conditional critical region is as follows:

```
var v : shared T;
    ...
region v do
   begin...await B ; ...end
```

In the example presented in Appendix II , variable v is a record consisting of two integer components which specify the number of readers and writers that currently use the resource.

Analysis of the use of conditional critical regions suggests the following observations. Critical regions have managed to structure the use of semaphores. This was achieved by both the explicit association of the data being shared with operations defined on them and the direct statement of the conditions under which computations are carried out. Problems still remained in this newer method of synchronization through shared variables. They are the following:

(i) Although the syncronization commands are now hidden in the statements "region v do" and "await B", the syncronization code

is still distributed through all the processes;

(ii) The repeated evaluation of the boolean expression B in the statement "await B" causes a considerable inefficiency in processing time.

In general, it can be said that critical regions are very inefficient whenever there is a heavy utilization of resources. In fact it can be said that it was the price paid for structuredness.

2.3 Monitors

The notion of good structure in parallel programming was re-assessed for the formulation of the concept of monitors (Hansen [5] and Hoare [21]). The concept was based on the contemporary notion of abstract data types (Hoare [20]). A monitor consists of a schedule composed of local data, procedures and functions, which is called by programs that need to acquire or release resources. Since the monitor is an abstract data type the programs that need to manipulate resources can only do it through the operations that define the type, without having any information about how the resources are implemented within the monitor.

The notation used for monitors was based on the class mechanism of SIMULA 67 [12]. In [21] monitors are described in the following manner:

```
monitor name : monitor
begin ... declarations of data local to the monitor;
   procedure procname (...formal parameters...);
      begin ... procedure body ... end;
      ... declarations of other procedures local to the monitor;
      ... initialization of local data of the monitor ...
end
```

Procedures of a monitor are called in the following way

```
monitor name - procname (...actual parameters...);
```

A monitor's procedures are common for all running processes. However, only one process can enter a monitor procedure at a time. The "wait" and "signal" operations must be preceeded by the name of the condition variable (e.g. card variable . wait and card variable . signal) since there may be, for instance, more than one reason for waiting. The solution to the readers and writers problem presented in Apprendix III is quoted from Hoare [21]. In the following example it should be noted that:

(i) There are four local procedures:

startread - entered by reader to initialize reading

endread - entered by reader when finishing reading

startwrite - entered by writer to initialize writing

endwrite - entered by writer when finishing writing

(ii) A conditioned function of the form "cardname - queue" returns the value true if any process is waiting for that cardname and returns false otherwise.

Monitors, as described, concentrated the code which was previously scattered through all the processes. Structuredness was achieved together with independence of the representation at a lower cost than that required by the repeated reevaluation of general boolean expressions in conditioned critical regions.

Some old problems were not solved. In fact, the centralization of the access to the resources via the monitor's procedures still relies on the user for the proper use of the procedures which are scattered through all the processes. If a user forgets to release some resource, he will block the whole subsystem controlled by the monitor.

Some new problems were introduced. The rule requiring that only one program at a time enter a monitor procedure may lead to excessive sequentialization of access to the resources. That will preclude, in some cases, a better utilization of processing time.

Given the short discussions about semaphores, critical regions and monitors above the key idea that we want to convey is that what is needed in parallel programming is not yet another extension of the semaphore concept which will necessarily still require the use of shared variables. One example of such an extension is the interesting

idea of path expressions [6,18]. Using path expressions it suffices to specify the order in which the operations on a shared object can be performed by different processes. A compiler will translate the path expression specification into the necessary semaphores and P and V statements. Path expressions provide a very high level language to deal with the synchronization problem but they are still based on the idea of shared variables and so share the shortcomings of the concepts discussed above.

What is presently needed is a major departure from the above programming style which will allow the development of reliable and efficient software for highly distributed systems. Such a programming technique based on processes and messages will be presented in the next section. It has already been discussed to some extent by the originators of the Thoth system [8,9].

There is a lesson that needs to be learned from the programming techniques based on synchronization via shared variables. The introduction of new programming constructs must be accompanied by the development of analytical tools which enable its rigorous characterization. Such tools consist of formal specification techniques and the corresponding verification methods.

3. Message Oriented Programming

In this section, we introduce the expression message oriented programming to refer to programming with processes and messages. This programming style has been adopted more or less systematically in the programming of parallel systems and has been called in the literature by different names. Programming through message passing [1], programming through managers [23] and proprietors [9] are some of the names used for more or less restricted versions of message oriented programming.

Message oriented programming has been proposed both at the theoretical level [38] and as a technique developed in connection with actual sytems implementations [23, 8]. To gain a wider acceptance it needs a precise characterization. In this section we attempt this characterization by stating some major program design principles associated with message oriented programming and by illustrating these principles through examples.

In message oriented programming, processes belonging to a given system of processes (which is also considered to be a process) communicate via messages that flow through the source-to-destination paths that connect them. Processes send messages through the communication paths and the messages are to be received at the destination some arbitrary finite time later. At the destination, all messages received are saved in arrival order in a separate queue for each of the sending processes. (Although this is the scheduling technique adopted, in

general many scheduling methods may be used in conjuction with message oriented programming). The structure of the system of processes determines the destination of the output stream of messages for the various processes. A message is removed from the queue when it is received by the corresponding process. We are assuming that imperfect communication -- messages arriving out of order or being lost -- is handled by a communication link process which verifies the communication protocol. (The order of the messages can be assured by attaching some sequence number mechanism and messages being lost can be avoided by a time-out mechanism.)

Message oriented programming sharply contrasts with parallel programming with synchronization through shared variables because the message model permits asynchronous communication only through message transmission with arbitrary delay and the asynchronous processes have disjoint state spaces.

The programming style of message oriented programming resembles the programming approach enforced by a highly extendable language (e.g. ECL [7]) which allows the definition by the user of both the control and data structures to be used in a given application. In fact, message oriented programming requires the program designer to provide both the definition of the process and message structures for a given application. In terms of problem solving two programming problems are present in this case: the question of how much of the program should be contained in the message structure and how much should be in the process structure, and the question to what extent should a program be data driven (that is, whether data should be dealt with more or less explicitly).

The inherent excessive flexibility of message oriented programming suggests that careful attention be paid to the methodological issues that it raises. Testing of these kinds of parallel programs is very difficult since, in general, it takes place in very general configurations of distributed machines (where the configuration may change dynamically).

3.1 Program Modularity

Message oriented programming responded to some of the very rigorous requirements for modularity proposed in the literature. In [13] a program segment can be called a module if it follows the properties of syntactic non-interference, semantic context independence and data generality. Syntactic non-interference accounts for the possibility of combining program segments without having to make syntactic changes in any of the segments. Semantic context independence insures that a given segment cannot cause side-effects and cannot be affected by side-effects. The property of data generality requires that modules be able to communicate via arbitrary data structures. Data generality allows for the full application of Parnas' "hiding principle" [34,35] through which each module's programmer needs to know only about another module's specification and not about its internal implementation.

The most difficult property to be satisfied for most programming systems is data generality. The very essence of message oriented programming, that is, the fact that communication between processes can only take place through message transmission and the fact that asynchronous processes have disjoint state spaces, satisfies it by definition.

The whole (process) structure of a message oriented program is itself specified by a process. Processes can be decomposed into an arbitrary number of component processes. In the programs to be presented later, we will use the symbols "{" and "}" to delimit processes in substitution to the usual pair begin / end. At execution time, the invocation of a process includes the invocation of all its constituent processes, since processes can invoke sub-processes as well as create and delete sub-processes.

The possibility of recursive process decomposition (nesting of processes) plus, once more, the mode used for process communication allow program segments in message oriented programming to be called modules, since they also satisfy the properties of syntactic non-interference and semantic context independence by definition.

3.2 Program Design

The design of the process and message structures that form a message oriented program must be guided by sound programming principles. These principles have in part been spelled out in the literature in connection with sequential programming and can almost directly be applied to the message oriented programming approach. Such principles are structured programming [15] , modular design [36,33] and programming with abstract data types [26].

From structured programming, the notion of program construction

by stepwise refinement can be directly used to structure processes as a hierarchy of virtual machines. The proper structuring of independent processes as a hierarchy of virtual machines not only facilitates the understanding of the system but also helps in defining which layers of the system are responsible for process creation and deletion and for the protection of resources. Stepwise refinement also characterizes the constructive approach of "writing the assertions before the code". This idea is readily applicable to message oriented programming.

The design of each main process in a parallel system must be preceded by the definition of a process control statement that specifies it. For instance, the specification of the control structure of the readers and writers problem can be stated rather simply. Let us define for this purpose the following three predicates:

- NRA - There is no reading activity
- NWA - There is no writing activity
- NWR - There is no writing request in the system

We recall that the reading and writing conditions of the problem can respectively be stated as:

- reading condition : $NWA \wedge NWR$
- writing condition : $NWA \wedge NRA$

Therefore, the process control statement for the readers and writers problem, expressed in our informal assertion language, is:

$$(NWA \wedge NWR) \vee (NWA \wedge NRA)$$

We will use the above statement later to informally prove the correctness of our example programs.

Modular design can give us a tool to deal with the problem of how much of the program should go in the process and message structures respectively. Myers [33] proposes two criteria to be used in decomposing systems into modules : module strength and module coupling. A balance between high strength and low coupling must be attempted. Module strength tries to achieve high module independence by maximizing the relationships within each module (and so minimizing dependence between separate modules). Minimizing module coupling is a process of both eliminating unnecessary relationships among modules and minimizing the tightness of those relationships that are necessary.

This procedure of achieving high strength and low coupling implies a clear identification of the resources needed in order to solve the problem. Our view of resource is an extension of the concept of abstract data type. In the commonly accepted view of abstract data types, there are no a priori constraints on the way that the operations from the type should be activated. These ideas agree with Myers' concept that a module should implement some data type in order to get high module strength and low module coupling. However, with modules corresponding to resources, in addition to the usual set of values and operations to be applied to them, we have to define how these operations are going to be synchronized (with each other and with the resource's external environment).

Motivated by the considerations given in the preceding paragraph, we postulate two ways of accomplishing the balance between high

strength and low coupling in message oriented programming: informational strength and functional strength. We will focus for the moment on informational strength. This criterion is used when the operations defined on the resource (or the associated abstract data type) are handled explicitly. A message oriented program has informational strength when each of its constituent processes performs a specific operation on the resource. The search for processes in this case can be done by determining all the asynchronous conditions in a problem (such as the process control statement given previously for the readers and writers problem) and associating to each of them a process that handles this particular aspect of the resource. (The conditions are separated in the process control statement by the alternative operator "or".)

If we look now to the reading and writing conditions in the process control statement for the readers and writers problem, the following process modularization can be stated to achieve high strength and low coupling using one process to handle each of the two stated asynchronous conditions ((NWA $\wedge$ NWR) and (NWA $\wedge$ NRA)). In Figure 1 the processes p-readers and p-writers follow the basic asynchronous conditions of the problem and the arrows indicate the source-to-destination paths that connect the processes. One message may flow through a given path at a given time. However, the same path may carry different values of messages at different times (indicated by the use of "or" below. The dotted area indicates which resource (named in capital letters) is being protected and the scope of its protection (i.e., which processes are in charge of it). In this case the access to the resource "FILE" is handled by the processes p-readers and p-writers. The resources readers and writers (enclosed in dotted

For the specific solution outlined in Figure 1, some operations are defined for message passing. A full definition of a data type intended for synchronization through message passing can be found in [11]. The following operations are some of the operations of this type:

(i) $send_i(j,msg)$: process i sends a message msg to process j ; the subscript may be omitted when it is clear which process is sending the message; no blocking is produced by this operation. When the value of msg is unimportant, i.e. when process i is just signalling process j , then we will abbreviate the construct to $send_i(j)$.

(ii) $receive_i(j)$: process i wants to receive a message from process j ; the receiving process blocks if there is no message in its buffer from the specified process.

(iii) $rec\text{-}any_i$: process i wants to receive a message from any of the running processes ; this operation returns a pair composed of the name of the process that sent the message and the message itself; the two components can be distinguished by var-name and var-msg ; in this case the receiving process blocks if there is no message from any of the other processes.

Note: another form of this operation could be $rec\text{-}any_i$(set-of-proc) where process i can receive a message from any process in the given

set. The first process in the set which is sending a message to the receiving process will satisfy this operation.

(iv) $istheremsg_i(j)$: This boolean function returns the value true if there is a message from process j to process i in the queue of messages and returns false otherwise.

Given the message structure informally specified above, we can express the first version of a message oriented program for the readers and writers problem. We will make use of the type "set" for which the operations of insertion and deletion of elements are defined. We also leave out of $send_i$, $receive_i$ and $istheremsg_i$ the subscript i when the value of i is obvious from the context.

```
p-reader( )
 {readercount : integer;
  setproc : set of strings;
  rproc : pair of strings;
  readercount := 0;
     while true do
     { if istheremsg(p-writer)
         then { rproc := receive(p-writer);
                while readercount ≠ 0 do
                {rproc := rec-any(setproc);
                readercount := readercount - 1;
                delete(rproc.name, setproc)}
                send(p-writer);
                receive(p-writer) }
```

```
        else { rproc := rec-any;
               if rproc.msg = startread
               then { send(rproc.name, OKtoread);
                      readercount := readercount + 1;
                      insert(rproc.name, setproc)}
               else { readercount := readercount - 1;
                      delete(rproc.name, setproc)}
               }
        }
}

p-writer( )
 {wproc : pair of strings ;
    while true do
        { wproc := rec-any ;
          send(p-reader) ;
          receive(p-reader) ;
          send(wproc.name, OKtowrite) ;
          receive(wproc.name) ;
          send(p-reader) }
}
```

We had proposed the following process control statement for the readers and writers problems: (NWA ∧ NWR) ∨ (NWA ∧ NRA). The following argument may be presented to show that the above assertion is satisfied:

(i) As soon as the process p-writer receives a request, it sends a signal to the process p-reader, and the test istheremsg(p-writer) assures that no further reading can take place. In this case the while loop enforces the completion of the previous readings before the process p-reader tells the process p-writer to go ahead.

(ii) The process p-writer controls the writing so that the process p-reader cannot proceed while there are some writing requests in the system. This assures mutual exclusion of reading and writing and it also provides priority for the action of writing. When there is no writing process waiting, more than one reading can be performed at the same time (by different invocations of the process p-reader).

The above very informal proof could be expressed rigorously if we had a formal specification for the program.

Another way to achieve high strength and low coupling is the use of the functional strength approach. Functional strength can be used when it is necessary or desirable to make the structure of the resource implicit. (The associated set of values and operations are treated implicitly and the operations are executed sequentially.) In other words, a functional strength solution packages together in one process all related operations. Whenever functional strength modules can be identified it is easier to develop a solution with lower coupling between modules since data management becomes centralized. By associating processes to resources to apply the above modularization criteria we are in essence adapting the concept of monitors to

message oriented programming. This technique has been called in the literature programming with managers [23] or proprietors [9].

Returning to the readers and writers example, we notice that it is possible to associate a single process to the resource FILE. A solution in which a process is dedicated to the protection of a resource, is outlined in Figure 2.

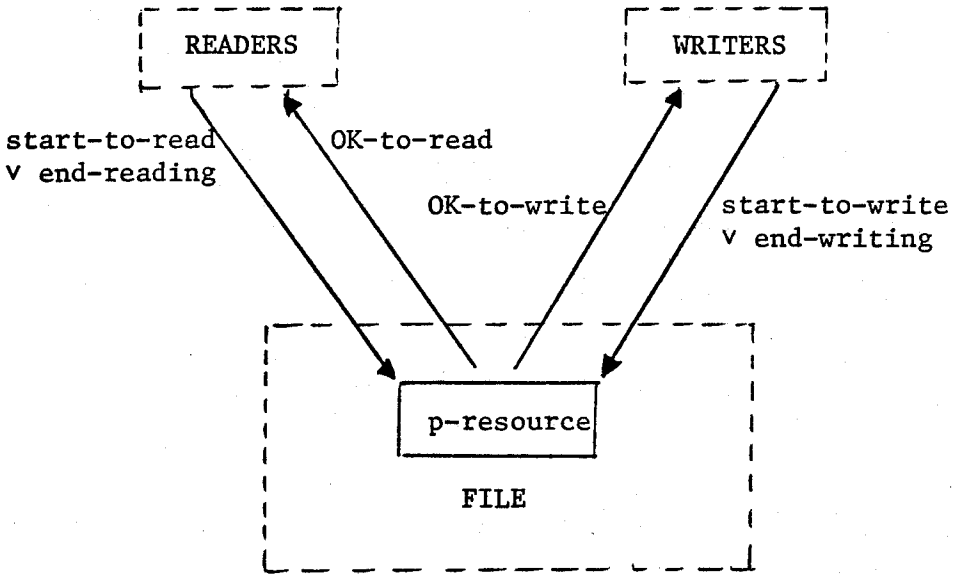

FIGURE 2

Such a data oriented programming style has been proposed in [23] and [9]. The general format of a resource-protecting process can be outlined as follows:

```
[manager]
[proprietor]  (  )

{  rec-any

     case 1
       .
       .
       .
     case n

}
```

Mutually exclusive modes of data manipulation.

It can be observed in the above process schema that if a different process is used for each case (as in the previous example) the need for communication links will grow and we will tend to have a higher coupling in the modular system. A solution for the readers and writers problem that follows the above approach can be expressed as follows:

```
p-resource( )
  { readercount : integer ;
    setproc : set of strings;
    mproc, mproc1 : pair of strings ;
    readercount := 0 ;
      while true do
      { if ¬istheremsgc(startwrite)
      then { mproc := rec-any ;
             if mproc.msg = startread
             then { send(mproc.name, OKtoread) ;
                    readercount := readercount + 1 ;
                    insert(mproc.name, setproc) }
             else { readercount := readercount - 1 ;
                    delete(mproc.name, setproc) }
      else  { mproc := receivec(startwrite) ;
              while readercount ≠ 0 do
              { mproc1 := receive(setproc) ;
                readercount := readercount - 1 ;
                delete(mproc1.name, setproc) }
                send(mproc.name, OKtowrite) ;
                receive(mproc.name) }
      }
}
```

Note that we have introduced two small changes in the message structure used in the above example. In fact we introduce into our communications data type the following operations:

(i) $istheremsgc_i(msg)$: asks if there is a message in the queue whose content is equal to msg.

(ii) $receivec_i(msg)$: process i wants to receive a message msg; the receiving process blocks if there is no message in its buffer whose content is equal to msg.

As before, the main process can be verified easily with respect to the same process control statement. The following informal argument can be used in this case:

(i) The test "$\neg$ istheremsgc(startwrite)" insures NWR and the end of any writing activity previously started (NWA) is guaranteed by the primitive "receive (mproc.name)" that blocks waiting for its occurrence. More than one reading can be performed at the same time.

(ii) As soon as a writing request arrives, the while loop forces the completion of all previous readings (NRA). The sequence (startwrite, endwrite) is activated without any external interference, thus assuring the termination of the writing activity before the next operation (NWA).

As the above program illustrated, the notion of data abstraction was introduced by having a process (or several processes) performing standard operations on the data and other processes wanting to use the data abstraction passing control to it. We could, as Jammel [23] has indicated, have serveral hardware processors running simultaneously within a data abstraction implemented as a message oriented program.

The above data driven approach to message oriented programming can be generalized by having not only one process (or several processes) dedicated to the protection of each resource involved in the problem but also by making these processes interact. A parallel programming system can then be expressed through communicating data abstractions. This programming approach leads to a very interesting structure pattern which can be applied to a number of situations. (The problem of protocols in computer networks is one of them.)

4. Specification of Message Oriented Programs

In the previous section we have described informally the distributed computing environment in which message oriented programs operate. We have also specified very informally all the programs that appeared in the text. That, of course, was responsible for the very informal proofs of correctness presented for these programs. It is universally accepted today that specification is a part of programming. Furthermore, as we are trying to characterize a new technique for parallel programming it is very important to state precisely the logical computing environment provided by a general purpose distributed computing system. This section is devoted to the presentation of the computation model that we assumed when writing our example programs and to the presentation of a specification technique for expressing and verifying properties of message oriented programs.

MacQueen [28] in an excellent recent survey studies some models for distributed computing. The work concentrates on message passing systems, for the same reasons that we decided to study programming techniques based on processes and messages. That is, the belief that the full promise of distributed computing is unlikely to be fulfilled unless a new programming technology is developed to match the new hardware systems.

MacQueen [28] classifies message passing systems according to the character of the communication medium. In models with direct communication, processes have global names and messages can be sent between two agents if and only if the source process knows the address

of the target process. In models with indirect communication, explicit channels or paths are used to transmit messages. We have assumed the form of direct communication in the programming environment used in the previous section. Milne and Milner [31] have proposed a model using channels in which there is information flow in one or both ways and multiple senders and receivers. In their model, operations are algebraic in character, operating independently of the internal structure of the processes to which they are applied. That allows for the direct application of structured programming as discussed before, since the composition operations on processes can be used at every level of the process hierarchy. The model also allows for the localization of the effects of explicit nondeterminism by considering all possible interleavings of a set of related events, therefore eliminating the need to deal with global interleavings of all events.

Since we implicitly followed Milne and Milner's model for providing the logical computing environment in which we wrote our programs we are going to start this section by describing this model. The description will be concise and based on the readers and writers example. The specification expressed in the language of the model will help in the understanding of the message oriented programs presented for the readers and writers problem.

The reservations which we have about this model can be stated as follows. The fact that Milne and Milner's model does not deal with the internal structure of processes makes it comparable to Petri nets and the other models described in MacQueen's paper [28]. In other words

the model is too far removed from the control and data structures of programs to guide the designer in constructing a process. One might compare it to the situation in which a sequential language is modelled semantically by using an SECD like mechanism (Landin [25]). Knowledge of this semantic model is of very little help in the construction of structured (sequential) programs.

To be able to bridge the gap between program specification and program implementation (expressed in the high level application language that we use), we resorted to a definitional specification technique. We show how the notion of algebraic specification can be formalized and used for both program design and verification of properties of parallel programs. We illustrate the method by using the preceding specification of the consumers and producers problem and by showing that the proposed solution is deadlock free.

4.1 The Flow Algebra Model of Distributed Computing Systems

In this section we present a concise description of the flow algebra model of distributed computing systems [31, 30] and apply it to an example.

The model sees a system of processes as a graph in which the nodes are processes represented by labelled ports and the edges are channels connecting these ports (and thus the processes to which the ports belong). The labelled ports used to represent processes are meant to capture the idea of the process's potential communication capabilities. To the model, the process is just a set of communication capabilities; that is, the communications (zero or more) which the

process offers to make with other processes at any given point in time. Each capability expresses not only the content of the offered communication but also the renewal (or continuation) process which will replace the communicating process following the communication. Suppose for example

$p = \{\alpha :: q_1, \alpha :: q_2, \gamma :: r\}$ is a set of communication capabilities. (A set is used since concurrency is reduced to the non-deterministic interleaving of all communication sequences). The possible renewals of process p are q_1, q_2 and r ; process p can carry out one of two α communications or a γ communication and then enter a renewal state which is q_1, q_2 or r respectively. The renewal (or state) which results from a communication activity is determined for different port names solely by the port being used for the communication. In the case of non-deterministic use of the same port name (as for α above), the renewal is chosen non-deterministically among the renewals specified for that port. As can be observed immediately, the model concentrates on the synchronization aspects of interprocess communication by abstracting the data communication aspects (only signals are modelled with no value passing being described).

The algebraic approach comes into play here because the behaviour of a composite process will be modelled by the composition of the behaviours of its sub-processes. In what follows we are going to present a specification for the readers and writers problem in the language of the model. We make use of a simplification of the scheduling technique presented in [32].

The operations read, request to write and write are labelled by the capabilities r, w_1 and w_2 respectively. Resource sharing processes are called $access_i$ and contain capabilities r, w_1 and w_2. The model introduces the idea of "controlled-access" to control the begin and the end of operations. Two functions, "before" and "after" are used to define the controlled-access.

Let α :: q belong to the set of capabilities of the process p. It means that if p communicates via α with another process (i.e., both processes have the same capability α) then q will follow p in the system. The function before(β', β) replaces all capabilities labelled by β, for example β :: q, in the capability set of a process p or its renewals by β' :: $\{\beta$ :: q$\}$. This indicates that a β communication may take place only after a β' communication.

The function after(β, β') replaces β :: q by β :: $\{\beta'$ :: q$\}$ to indicate that a β communication has occurred and is followed by a β communication.

In the following definition of $controlled\text{-}access_i$ the following abbreviations are used: rd for begin reading, wt_1 for begin writing request, wt_2 for begin writing, rf for end reading and wf for end writing. The dot stands for function composition.

$$Controlled\text{-}access_i = before(rd, r).after(r, rf).before(wt_1, w_1).before(wt_2, w_2).after(w_2, wf)(access_i)$$

The scheduler s , which performs the control for the readers and writers problem is defined recursively as follows:

i. $s = s_1(0)$

ii. $s_1(0) = \{wt_1 :: s_2(0), rd :: s_1(1)\}$

iii. $s_1(i) = \{wt_1 :: s_2(i), rd :: s_1(i+1), rf :: s_1(i-1)\}$ for $i \geq 1$

iv. $s_2(0) = \{wt_2 :: \{wf :: s\}\}$

v. $s_2(i) = \{rf :: s_2(i-1)\}$ for $i \geq 1$.

The system as a whole is described by

$$system = (controlled\text{-}access_1 \| \ldots \| controlled\ access_n) \| s$$

where $\|$ stands for a derived operation which is applied whenever two composite processes require their interconnected parts to be invisible to other processes. (The basic operations in the theory are composition, restriction of the set of ports of a given process and relabelling of processes).

In what follows we give the intuitive interpretation of the definition of the scheduler s above:

i. defines the initial state of s

ii. defines a state in which a write operation is performed but where there is no read operation or writing request.

iii. states that some reading operations are being performed

iv. states that there is a writing operation being executed

v. states that when there is a writing request, the process makes sure that all currently active readers finish before the writing operation is initiated.

Note that the model does not make use of a queue of messages in the communication system. If a writing operation is being processed then all other possible operations are delayed until the end of the writing operation.

The behavior of the specification of the readers and writers problem as expressed in terms of the flow algebra model can be summarized in the following diagram that represents the scheduler s (for $i \geq 1$)

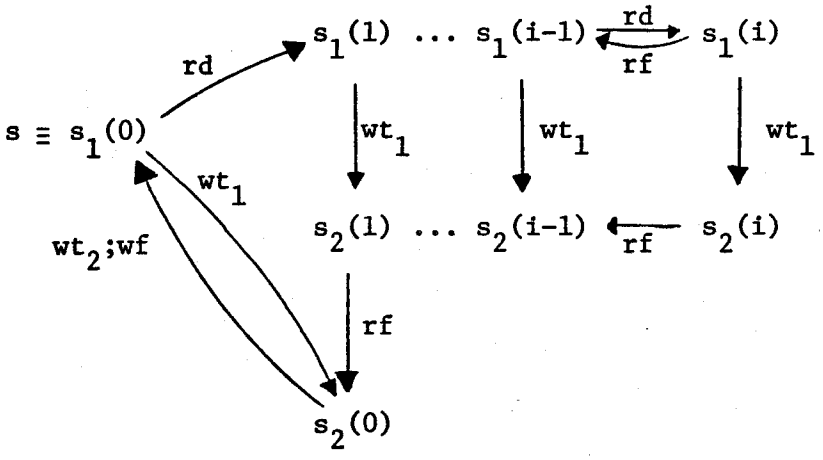

Figure 3

As Figure 3 illustrates, the flow algebra model is in essence an operational model characterized by states and transitions. It bears similarities to automata models, but the operations it defines provide a richer structure that cannot be found in previous models.

4.2 Algebraic Specification for Process Communication

The flow algebra model helped to further characterize the computing environment in which message oriented programs operate. The reason we used it to specify the readers and writers problem was to emphasize the conceptual distance that exists between this level of specification and the programming level presented in the previous section. For that matter, the flow algebra model is representative of the class of models surveyed by MacQueen in [28], which have proved to be useful in the specification and analysis of parallel system but which fail to assist programmers in the design or natural verification of parallel programs.

To be able to make the specification level adhere more naturally to the programming level we are going to adopt a definitional specification technique. Following Liskov's classification [27], definitional methods include axiomatic and algebraic specifications. We are in the process of extending the algebraic specification method, to use it for the specification of parallel programs.

We are now going to introduce the algebraic approach for the specification of parallel programs and illustrate its use for verification by addressing a central problem found in message oriented programming: the deadlock problem. The ease of proving the deadlock freeness property in the consumers-producers example will serve the purpose of illustrating the power of the algebraic specification approach. It is, at the present, preferred to the axiomatic approach because of its ability to abstract from the

structural details contained in the problem solution.

Now we present a solution for the producer-consumer problem that is deadlock free in order to illustrate the use of our approach. The problem can be stated informally in the following way: a producer and a consumer process interact by means of a buffer area into which the producer "deposits items" and from which the consumer "extracts items"; the two processes repeat their actions continuously and it is known that the buffer is large enough to hold n items. It is possible to base the solution of the problem on two variables or resources: avpl (number of available places in the buffer) and avit (number of available items in the buffer). To each of these two resources will be associated a process which will be responsible for controlling access to it. Note that we are assuming the use of an implicit buffer in this solution.

```
producer( )
   { last : pointer;
     x : pair of strings;
        while true do
        { produce message;
          x.msg := wait;
          while x.msg = wait do
          { send(p-avpl);
            x := receive(p-avpl)}
          place message at last;
          last := last + 1(mod n);
          send(p-avit)
        }
   }
```

```
consumer( )
   { first : pointer;
     y : pair of strings;
        while true do
        { y.msg := wait;
          while y.msg = wait do
          { send(p-avit);
          y := receive(p-avit)}
          get message from first;
          first := first + 1(mod n);
          send(p-avpl);
          consume message
        }
}
```

```
p-avpl( )
   { avpl : integer;
     t : pair of strings;
        avpl := n;
        while true do
        { t := rec-any;
          if t.prc = producer
          then if avpl = 0 then send(producer, wait)
                   else { avpl := avpl - 1;
                          send(producer, goahead)}
          else avpl := avpl + 1}
    }
```

```
p-avit( )
   { avit : integer;
     u : pair of strings;
     avit := 0;
       while true do
       { u := rec-any;
         if u.prc = consumer
         then if avit = 0 then send(consumer, wait)
                  else { avit := avit - 1;
                         send(consumer, goahead)}
         else avit := avit + 1}
    }
```

In our proposed solution to the producer and consumer problem a process can be in a deadlock situation if it is blocked while executing a receive operation. Thus, the condition for the occurrence of a deadlock in the system is the existence of a circular chain of processes in which each process is blocked and is waiting for a message **from the next process in the chain.**

The asynchronous nature of message oriented programming determines the possible occurrence of delay situations such as (i) processes sending message before they can actually be processed; (ii) processes trying to receive messages before they are sent. The idea we introduce to handle delays is what we call a canonical synchronization formula (csf). This formula is a canonical representation of the sequence of all communication operations ("sends" and "receives") performed by a parallel program. In the csf, the overall blocking (delay) period between send operations and the corresponding receive operations is minimized. Intuitively, what the

formula means is that a message sent by process p_1 is received as soon as possible by the target process p_2 (immediately in a well designed system). Thus, a csf is a specification tool which attempts to specify the intended history of communications actions in the system. A given execution of the system will not in general result in the same sequence of communications actions but it is intended that whatever sequence results, it be equivalent to the sequence specified by the csf. It is of course necessary to prove that the non-determinism introduced by the asynchronous behavior of the system does not affect the result (and so the csf does describe the behavior of the system).

We use what we call "synchronization axioms" which transform a given sequence of communication primitives into the corresponding csf. This is done by looking at the delay situations. If the sending operation is performed before it can be processed (i.e. the sending and receiving operations are not together) then this "send" can commute with the next primitive in the expression. The same happens if there is a receiving operation that was performed before the message was sent. There is no change of places where the receiving operation is preceded by the corresponding sending operation. It is, of course, clear that primitives of the same process cannot be interchanged; otherwise the original order determined by the sequential program would be violated.

Let us assume a set of synchronization formulae which describes the communication aspect of a parallel program. These expressions (sf's) give the possible communication skeletons for each of the processes that compose the program (by relating the different communication behavior of a process). The set of communication behaviors (subexpressionsseparated

by the operator or) form the corresponding sf. By grouping the related subexpressions - i.e. the ones that refer to each other and are activated together -- we can identify distinct behaviors of the program. Instead of doing a general shuffle of the sf's, we will derive directly the csf for each of the groups of interrelated subexpressions. A message oriented program is deadlock free if we can construct a csf for each of the groups of interrelated subexpressions. Otherwise, we may have cases of potential deadlock or an unavoidable deadlock situation.

If we consider now the solution proposed to the consumer and producer problem, we can apply the proposed technique for deadlock detection. The first step is to derive the several synchronization formulae from the code through which each process is expressed in the program. The expressions for the four processes used in the program are given below. Let us denote send by s, receive by r, producer by pd, consumer by cs, p-avpl by pl and p-avit by it in the following expressions. The symbol ";" denotes sequentiality of actions (as in programs).

1. **Producer:**

   $[(s(pl);\ r(pl))^{i};\ s(it)]$ for $i \in N$

2. Consumer:

   $[(s(it);\ r(it))^{j};\ s(pl)]$ for $j \in N$.

3. Proprietor of resource avpl:

   $[\underbrace{(r(pd);\ s(pd, w))^{i-1};\ (r(pd);\ s(pd, go))}_{3a}\ \underline{or}\ \underbrace{(r(cs))}_{3b}]$

   for $i \in N$. "or" denotes that more than one **expression can be used for the process.**

The two alternative expressions in this case arise from the use of rec-any in the program defining avpl. The possible senders for this rec-any are either the producer or the consumer. Thus, the rec-any operation can be replaced by the equivalent receive(producer) (r(pd)) and receive(consumer) (r(cs)) operations, respectively. The two possibilities given rise to different sequences of actions thus giving rise to expressions 3a and 3b, respectively.

4. Proprietor of resource avit:

$$\underbrace{[(r(cs);\ s(cs,\ w))^{j-1};\ (r(cs);\ s(cs,\ go))}_{4a}\ \underline{or}\ \underbrace{(r(pd))}_{4b}]$$

for $j \in N$. The alternative expressions arise for reasons analogous to those in 3.

A simple analysis of the program code allowed us to derive the i and j exponents used in the expressions 1 to 4 above. The analysis consisted of finding matching subsequences of sends and receives in the computation sequences of the interacting processes. Exponent i in expression 1 expresses the fact that producer received from the proprietor of resource avpl a wait message (called simply w above) (i-1) times before it was permitted to proceed (or go). Therefore the first part of expression 1 (corresponding to exponent i) is related to the first alternative used in expression 3 (called 3a). It can also be seen that the last part of the first expression (s(it)) corresponds to the second sub-expression of the fourth expression (r(pd)) . Symmetrically, the the exponent j used in expression 2 can be related to the corresponding alternatives in expressions 3 and 4.

Since the logical end of the various processes is located at the physical end of their codes, the expression that corresponds to the execution of each of the processes is the transitive closure (repetition) of the expressions 1 to 4 given above.

By analysing the code of the program we find that the expression 1 to 4 can be divided into two interrelated groups: firstly 1, 3a, 4b and secondly 2, 3b, 4a. If we manage to construct the two csf's that describe the joint behavior of the alternatives that form each of the distinct groups, then the program is deadlock free. We match the send commands with the corresponding receive commands to construct directly the canonical synchronization formula for the two alternatives. We give below the csf's for the groups of alternatives (1, 3a and 4b) and (2, 3b and 4a).

1. Processes pd, pl and it:

$$\prod_{i=1}^{m} [((s_{pd}(pl);\ r_{pl}(pd);\ s_{pl}(pd,w);\ r_{pd}(pl))^{n_i-1};$$
$$(s_{pd}(pl);\ r_{pl}(pd);\ s_{pl}(pd,go);\ r_{pd}(pl));\ (s_{pd}(it);\ r_{it}(pd)))]$$

for $n_i \in N$. We use the symbol $\prod$ to denote concatenation of expressions.

2. Processes cs, it and pl:

$$\prod_{i=1}^{m'} [((s_{cs}(it);\ r_{it}(cs);\ s_{it}(cs,w);\ r_{cs}(it))^{n_i'-1};$$
$$(s_{cs}(it);\ r_{it}(cs);\ s_{it}(cs,go);\ r_{cs}(it));\ (s_{cs}(pl);\ r_{pl}(cs)))]$$

for $n_i' \in N$.

By matching the send commands with the receive commands we constructed the csf's corresponding to the two groups of expressions above. (We have skipped the proof that the csf's are a canonical representation of the general shuffle of these groups of expressions.) We may then conclude that there is no process blocked forever while performing a receive

operation because there are no unpaired receive operations and there is no deadlock because we were able to construct both csf's.

5. Conclusions

The objective of the present paper was to characterize more precisely a new programming style which we called message oriented programming. This programming style has been suggested in general terms in both theory and practice. Zave [38, 39] showed that inter-process communication via message passing is more powerful than synchronization through shared variables and McQueen [28] has surveyed models of computation which describe abstractly the semantics of the message passing mechanism. Some practical efforts have illustrated the power of the method while hinting at some programming practices which proved useful during its application to the implementation of real systems [23, 8].

Our emphasis was placed on the presentation of the method through the statement of the programming principles on which it is based and the proposal of a specification technique which can naturally be associated to it. Examples were used to illustrate most of the ideas.

We do not propose a programming language for message oriented programming nor do we give a set of rules for deriving message oriented programs from their specifications. Much inter-disciplinary work is needed in these directions. Efforts must be undertaken to relate the design principles of message oriented programming to the design of fully distributed systems and to the development of techniques for the specification and verification of properties of such programs. The initial results of the latter

effort have been illustrated in the use of a simple communications data type in the development of processes and the verification of their properties. This data type and the relevant verification techniques are more fully specified in [11].

Acknowledgements

The authors would like to thank the referees for their careful and constructive comments.

A P P E N D I X I

```
integer readcount, writecount; (initial value = 0)
semaphore mutex1, mutex2, mutex3, r, w; (initial value =1)
comment READER;
begin
P(mutex3);
   P(r);
      P(mutex1);
      readcount := readcount + 1;
      if readcount = 1 then P(w);
      V(mutex1);
   V(r);
V(mutex3);
   . . .
   reading is done
   . . .
P(mutex1);
   readcount := readcount - 1;
   if readcount = 0 then V(w);
V(mutex1)
end
comment WRITER;
begin
P(mutex2);
   writecount := writecount + 1;
   if writecount = 1 then P(r);
V(mutex2);
P(w);
   . . .
   writing is done
   . . .
V(w);
P(mutex2);
   writecount := writecount - 1;
   if writecount = 0 then V(r);
V(mutex2);
end
```

A P P E N D I X I I

```
var v : shared record readers, writers : integer end (initial value = 0)
    w : shared boolean;
          . . .
comment READER;
begin
   region v do
   begin await writers = 0;
      readers := readers + 1 end

          . . . read . . .;

   region v do
      readers := readers - 1
end
   comment WRITER;
begin
   region v do
   begin writers := writers + 1;
      await readers = 0 end
   region w do . . . write . . .;
   region v do
      writers := writers - 1
end
```

A P P E N D I X I I I

```
class readers and writers : monitor
begin readercount : integer;
   busy : boolean;
   OKtoread, OKtowrite : condition;
   procedure startread;
   begin if busy v OKtowrite.queue then OKtoread.wait;
      readercount := readercount + 1;
      OKtoread.signal;
      comment once one reader start, they all can
   end startread;
   procedure endread;
   begin readercount := readercount - 1;
      if readercount = 0 then OKtowrite.signal
   end endread;
   procedure startwrite;
   begin
      if readercount ≠ 0 v busy then OKtowrite.wait;
      busy := true
   end startwrite;
   procedure endwrite;
   begin busy := false;
      if OKtoread.queue then OKtoread.signal
                        else OKtowrite.signal
   end endwrite;
   readercount := 0 ;
   busy := false
end readers and writers;
```

References

[1] Baskett, F., Howard, J.H. Montague, J.T. : Task Communication in DEMOS; Proceedings of the 6th ACM Symposium on O.S. Principles, 1977.

[2] Brinch Hansen, P. : The Nucleus of an Operating System; CACM, April 1970 (pp. 238-241, 250).

[3] Brinch Hansen, P.: A Comparison for Two Synchronizing Concepts; Acta Informatica 1, 1972 (pp. 190-199).

[4] Brinch Hansen, P.: Structured Multi-Programming; CACM, July 1972 (pp. 574-578).

[5] Brinch Hansen, P. : Operating System Principles; Prentice-Hall, Englewood Cliffs, N.J., 1973.

[6] Campbell, R.H., Habermann, A.N. : The Specification of Process Synchronization by Path Expressions; Lecture Notes in Computer Science, Springer-Verlag, Vol. 16, 1974.

[7] Cheatham, T.E. : The Reeent Evolution of Programming Languages; IFIP 71, Ed. C. Freiman, Vol. 1 (pp. 289-313).

[8] Cheriton, D.R., Malcolm, M.A., Melen, L.S., Sager, G.R. : Thoth, A Portable Real-Time Operating System; CACM, February 1979.

[9] Cheriton, D.R. : Multi-Process Structuring and the Thoth Operating System; Ph.D. Thesis, University of Waterloo, August, 1978.

[10] Courtois, P.J., Heymans, F., Parnas, D.L. : Concurrent Control with "Readers" and "Writers"; CACM, October 1971 (pp. 667-668).

[11] Cunha, P.R.F., Maibaum, T.S.E. : A Communications Data Type for Message Oriented Programming; to be presented at the 4 éme Colloque International sur la Programmation, Paris, April 1980.

[12] Dahl, O.J. : Hierarchical Program Structures; in Structured Programming, Academic Press, N.Y., 1972.

[13] Dennis, J.B. : Modularity; in Advanced Course on Software Engineering, Ed. F. Bauer, Springer-Verlag, 1973.

[14] Dijkstra, E.W. : Cooperating Sequential Processes; Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968, (pp. 43-112).

[15] Dijkstra, E.W. : Notes on Structured Programming, Structural Programming, Academic Press, London, 1972.

[16] Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.F. : An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types; IBM Research Report RC 6487, 1976.

[17] Guttag, J. : The Specification and Application to Programming of Abstract Data Types; Ph.D. Thesis, CSRG TR 59, University of Toronto, Sept. 1975.

[18] Habermann, A.N. : On the Concurrency of Parallel Processes; Perspectives on Computer Science, A. Jones (ed.), Academic Press, London, 1977 (pp. 77-90).

[19] Hoare, C.A.R. : Towards a Theory of Parallel Programming; International Seminar on O.S. Techniques, Belfast, Northern Ireland, August-September 1971.

[20] Hoare, C.A.R. : Proof of Correctness of Data Representation; Acta Informatica 1, 1972 (pp. 271-281).

[21] Hoare, C.A.R. : Monitors, an Operating System Structuring Concept; CACM, October 1974 (pp. 549-557).

[22] Hoare, C.A.R. : Communicating Sequential Processes; CACM August 1978 (pp. 666-677).

[23] Jammel, A.J., Stiegler, H.G. : Managers versus Monitors; Proceedings of the IFIP 1977 (pp. 827-830).

[24] Keller, R.M. : Formal Verification of Parallel Programs; CACM, July 1976 (pp. 371-385).

[25] Landin, P.J. : The Mechanical Evaluation of Expressions; Computer Journal Vol. 6, No. 4, (pp. 308-320), 1964.

[26] Liskov, B.H., Zilles, S.: Programming with Abstract Data Types; Proc. Conf. on Very High Level Languages, SIGPLAN, Vol. 9, April 1974.

[27] Liskov, B.H., Zilles, S. : Specification for Data Abstractions; IEEE on S.E., Vol. SE-1, March 1975.

[28] MacQueen, D.B. : Models for Distributed Computing; Proc. of EEC/IRIA Course on the Design of Distributed Processing, Nice, France, July 1978.

[29] Manning, E.G., Peebles, R.W. : A Homogeneous Network for Data-Sharing Communications; Computer Networks 1, 1977 (pp. 211-224).

[30] Milne, G. : A Mathematical Model of Concurrent Computation; Ph.D. Thesis, University of Edinburgh, CST-4-78, March 1978.

[31] Milne, G., Milner, R. : Concurrent Processes and their Syntax; JACM, Vol. 26, No. 2, April 1979.

[32] Milne, G. : Scheduling within a Process Model of Computation; 1st European Conf. on Parallel and Distributed Processing, Toulouse, France, 1979.

[33] Myers, G.J. : Composite/Structured Design; van Nostrand Reinhold Co., 1978.

[34] Parnas, D.L. : Some Conclusions from an Experiment in Software Engineering, Proc. of the 1972 FJCC.

[35] Parnas, D.J. : A Technique for Software Module Specification with Examples; CACM, May 1972 (pp. 330-336).

[36] Parnas, D.J. : On the Criteria to be Used in Decomposing Systems into Modules; CACM, December 1972 (pp. 1053-1058).

[37] Vantilborgh, H., van Lawsweerde, A.: On an Extension of Dijkstra's Semophore Primitives; IPL, Vol. 1, 1972 (pp. 181-186).

[38] Zave, P. : On the Formal Definition of Processes; Conf. on Parallel Processing, Wayne State University, IEEE Computer Society, 1976.

[39] Zave, P. : A Design Tool for Real-Time Processes; Conf. on Information Sciences and Systems, Johns Hopkins University, 1977.

[40] Zilles, S.N. : Algebraic Specification of Data Types; Proj. MAC Report 11, MIT, Cambridge, Mass., 1974 (pp. 25-28).