A COMMUNICATION DATA TYPE FOR MESSAGE ORIENTED PROGRAMMING[†]

P.R.F. Cunha and T.S.E. Maibaum*

Abstract: In this report we resort to a definitional specification technique in order to be able to bridge the gap between program specification and program implementation (expressed in a high level application language) in message oriented programming. We show how the notion of algebraic specification can be used to formalize the communications primitives and then applied to the verification of communications properties of parallel programs. We illustrate the method by introducing elements of a calculus for reasoning about message passing programs and then using a specification of the consumer and producer problem and showing that the proposed solution is deadlock free.

[†] This work was partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

* University of Waterloo, Department of Computer Science, Waterloo, Ontario, CANADA N2L 3G1.

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. Process communication and synchronization can be achieved either through shared variables or by message transmission. It has been shown [22] 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. 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[22], Jammel[14], Hoare[13] and in the description of multiprocessing systems such as Demos[1], Mininet[19] and Thoth[4,5].

Zave[22] 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 technique 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.

MacQueen[18] 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 models 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.

The reservations which we have about the models surveyed in [18] can be stated as follows. The fact that the flow algebra model of [21], the actor model of [11], and the stream (data flow) model of [15] do not deal with the internal structure of processes make them, like Petri nets, very operational in nature. 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[16]). 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 a high level application language), we resort to a definitional specification technique. We show in section 2 how the notation of algebraic specification [9,10,17] can be formalized and used for modelling of properties of parallel programs. We illustrate the method in section 3 by outlining the elements of a calculus for reasoning about message passing programs and then using a specification of the consumer and producer problem and showing that the proposed solution is deadlock free.

2. A MODEL FOR MESSAGE PASSING

Let us assume that in a certain configuration of a system we have n processes. Conceptually, to each of them we associate a set of n different buffers, which can just contain one message individually. Each buffer is labelled with one of the process names of the system and thus can only receive messages from this particular process. One possible conceptual representation for this communication model would be a square matrix where the rows and columns would be labelled with each of the process names. Every time a create process operation is executed, we increase both the number of rows and columns by one and we decrease them by the same value in the destroy operation. We use the idea of environment in order to capture all the past history of the communication mechanism, recording how the matrix, referred to above, evolves from one configuration of values to another. The concept of environment models the interrelationship between processes which is altered only by the communication mechanism. A practical implementation is present in the construct called "switch" that is implemented in communications networks [19].

We will be considering the usual communication primitives found in most parallel systems: send, receive, create and destroy. It is assumed the non-blocking send and blocking receive are used although we can adequately simulate the other possibilities such as blocking send and blocking receive and so on. Note that we have not specified the body of the process in the operation create. (We preferred a more abstract definition to concentrate only on the effect of various primitives on the communication environment. We do not deal with issues raised by the nature of a host language - such as which processes can create and destroy and how this can be done.) We also define basic operations such as istherepro (is there a process), istheremsg (is there a message) and isprcblocked (is the process blocked). The only way that a process can block is to do a receive operation on an empty buffer.

We are now ready to define our communications data type. The sorts used are Environment (or Env), Process-name, Message, and Boolean (or Bool). The syntax of the operations is defined in the style of [9] as follows:

1. ϕ: $\rightarrow$ Env
2. create: Process-name $\times$ Env $\rightarrow$ Env
3. destroy: Process-name $\times$ Env $\rightarrow$ Env
4. send_i: Process-name $\times$ Message $\times$ Env $\rightarrow$ Env for each $i \in$Process-name (P_n)
5. receive_i: Process-name $\times$ Env $\times$ Message $\rightarrow$ Env for each $i \in P_n$
6. istherepr c: Process-name $\times$ Env $\rightarrow$ Bool
7. istheremsg: Process-name $\times$ Process-name $\times$ Env $\rightarrow$ Bool
8. isprcblocked: Process-name $\times$ Env $\rightarrow$ Bool
9. isprcblocked: Process-name $\times$ Process-name $\times$ Env $\rightarrow$ Bool
10. x: Process-name $\times$ Process-name $\times$ Env $\rightarrow$ Message
11. i: $\rightarrow$ Process-name for each $i \in P_n$
12. msg: $\rightarrow$ Message for each msg $\in$ Message $\cup$ {no-msg}
13. error: $\rightarrow$ s for each $s \in$Sorts

We now state the axioms for the type. Consider cp $\in$ {create(ℓ), destroy(ℓ), send_ℓ(m,msg), receive_ℓ(m)} for ℓ,m variables ranging over Process-name, σ and σ' ranging over Env, msg ranging over Message.

1. create(ℓ)(cp(σ)) = create(ℓ)(σ') =

$$= \begin{cases} \text{error} & \underline{\text{if}}\ \text{cp eq create}(\ell) \\ \text{create}(\ell)(\text{cp}(\sigma)) & \underline{\text{if}}\ \sigma'\ \text{eq}\ \phi \\ \text{cp}(\text{create}(\ell)(\sigma)) & \text{otherwise} \end{cases}$$

where the symbol "eq" denotes syntactic equality between terms.

2. destroy(ℓ)(σ) = $\underline{\text{if}}$ $\neg$ isthereprc(ℓ)(σ) $\underline{\text{then}}$ error
3. send_ℓ(m,msg)(σ)

$$= \underline{\text{if}}\ (\neg\ \text{isthereprc}(\ell,m)(\sigma) \vee \text{istheremsg}(\ell,m)(\sigma))\ \underline{\text{then}}\ \text{error}$$

where $\neg$isthereprc(ℓ,m)(σ) $\equiv$ $\neg$isthereprc(ℓ)(σ) $\vee$ $\neg$isthereprc(m)(σ)

4. receive_ℓ(m)(σ)

$$= \underline{\text{if}}\ (\neg\ \text{isthereprc}(\ell,m)(\sigma))\ \underline{\text{then}}\ \text{error}$$

5. send_ℓ(m,msg)(σ) = $\underline{\text{if}}$ isprcblocked(ℓ)(σ) $\underline{\text{then}}$ error
   receive_ℓ(m)(σ) = $\underline{\text{if}}$ isprcblocked(ℓ)(σ) $\underline{\text{then}}$ error
6. x(ℓ,m)(cp(σ)) = x(ℓ,m)(σ') =

$$= \begin{cases} \text{error} & \underline{\text{if}}\ \neg\text{isthereprc}(\ell,m)(\sigma') \\ \text{msg} & \underline{\text{if}}\ (\text{cp eq send}_m(\ell,\text{msg}) \wedge \neg\text{isprcblocked}(\ell,m)(\sigma) \\ \text{no-msg} & \underline{\text{if}}\ ((\text{cp eq send}_m(\ell,\text{msg}) \wedge \text{isprcblocked}(\ell,m)(\sigma)) \vee \\ & \quad \text{cp eq receive}_\ell(m) \vee \text{cp eq create}(\ell) \vee \\ & \quad \text{cp eq create}(m)) \\ x(\ell,m)(\sigma) & \text{otherwise} \end{cases}$$

7. istheremsg(ℓ,m)(σ)

$$= \begin{cases} \text{error}\ \underline{\text{if}}\ x(\ell,m)(\sigma)\ \text{eq error} \\ \underline{\text{false}}\ \underline{\text{if}}\ x(\ell,m)(\sigma)\ \text{eq no-msg} \\ \underline{\text{true}}\ \text{otherwise} \end{cases}$$

8. $\text{isprcblocked}(\ell)(\text{cp}(\sigma)) - \text{isprcblocked}(\ell)(\sigma') =$

$$= \begin{cases} \text{error } \underline{\text{if}} \neg \text{isthereprc}(\ell)(\sigma') \\ \underline{\text{true}}\ \underline{\text{if}}\ (\text{cp eq receive}_\ell(m) \wedge \neg \text{istheremsg}(\ell,m)(\sigma)) \\ \underline{\text{false}}\ \underline{\text{if}}\ (\text{cp eq create}(\ell) \vee \text{cp eq send}_\ell(m,msg) \vee \\ \qquad (\text{cp eq receive}_\ell(m) \wedge \text{istheremsg}(\sigma,m)(\sigma)) \\ \text{isprcblocked}(\ell)(\sigma) \text{ otherwise} \end{cases}$$

9. $\text{isprcblocked}(\ell,m)(\text{cp}(\sigma)) = \text{isprcblocked}(\ell,m)(\sigma') =$

$$= \begin{cases} \text{error } \underline{\text{if}} \neg \text{isthereprc}(\ell,m)(\sigma') \\ \underline{\text{true}}\ \underline{\text{if}}\ (\text{cp eq receive}_\ell(m) \wedge \neg \text{istheremsg}(\ell,m)(\sigma)) \\ \underline{\text{false}}\ \underline{\text{if}}\ (\text{cp eq create}(\ell) \vee \text{cp eq create}(m) \vee \\ \qquad \text{cp eq send}_\ell(m,msg) \vee (\text{cp eq receive}_\ell(m) \wedge \\ \qquad \text{istheremsg}(\ell,m)(\sigma))) \\ \text{isprcblocked}(\ell,m)(\sigma) \text{ otherwise} \end{cases}$$

10. $\text{isthereprc}(\ell)(\text{cp}(\sigma)) = \text{isthereprc}(\ell)(\sigma') =$

$$= \begin{cases} \text{error} & \underline{\text{if}}\ \sigma' \text{ eq} \\ \underline{\text{true}} & \underline{\text{if}}\ \text{cp eq create}(\ell) \\ \underline{\text{false}} & \underline{\text{if}}\ \text{cp eq destroy}(\ell) \\ \text{isthereprc}(\ell)(\sigma) & \text{otherwise} \end{cases}$$

[Note the way the environment σ is used in the axioms (i.e. not as specified by the syntax). This notation is preferred because sometimes the environment will be implicit, as for example, in the code of a program.]

We will now endeavour to explain the intutitive meanings of (some of) these axioms:

1. The creating operation cannot create a process with a name that was used to label another process previously. This procedure implies that a name can just be taken once.

3. In this model one process cannot send two consecutive messages to another one; it has to wait until the first message is received. Therefore, it is "error" if the receiving process m does not exist or if process ℓ tries to superpose a a message on m's buffer.

4. Performing a receive operation when either the sending or receiving process does not exist results in error.

7. $\text{istheremsg}(\ell,m)(\sigma)$ determines if, in the present environment, there is a message sent by process ℓ to process m which has not yet been received by process m. The operation $x(\ell,m)(\sigma)$ determines the value of the switch for the processes ℓ and m in the present environment by examining communication primitives used in building the present environment.

8. A process ℓ is blocked in an environment σ' if it is trying to do a receive operation for which there has not yet been a corresponding send. So the axiom

determines if ℓ is blocked in σ' as follows:

(i) If there is no process ℓ in σ', then we have error;

(ii) If σ' = $\text{receive}_\ell(m)(\sigma)$ and there is no message from m to ℓ in σ, then true;

(iii) If σ' = create(ℓ)(σ) then false (since ℓ cannot be blocked if it has just been created). If σ' = $\text{send}_\ell(m,\text{msg})(\sigma)$ then false since ℓ cannot be blocked if it has just done a send. (Finally, it is also false if we have the negation of condition (ii);)

(iv) Otherwise, we can determine the value of isprcblocked(ℓ)(cp(σ)) by evaluating isprcblocked(ℓ)(σ). That is, we look in the previous environment.

10. The explanation of this axiom is analogous to the previous one. We compute if process ℓ exists in environment σ' by looking for create(ℓ) and destroy(ℓ) in the communication primitives used to build σ'.

Let us look at a simple example:

ϕ	$\equiv \sigma_0$;
create(1)(σ_0)	$\equiv \sigma_1$;
create(2)(σ_1)	$\equiv \sigma_2$;
$\text{send}_1(2,\text{msg}_1)(\sigma_2)$	$\equiv \sigma_3$;
create(3)(σ_3)	$\equiv \sigma_4$;
$\text{send}_3(2,\text{msg}_2)(\sigma_4)$	$\equiv \sigma_5$;
destroy(3)(σ_5)	$\equiv \sigma_6$;
$\text{receive}_2(3)(\sigma_6)$	$\equiv \sigma_7$;
create(4)(σ_7)	$\equiv \sigma_8$;
$\text{send}_4(1,\text{msg}_3)(\sigma_8)$	$\equiv \sigma_9$;
$\text{send}_2(4,\text{msg}_4)(\sigma_9)$	$\equiv \sigma_{10}$;

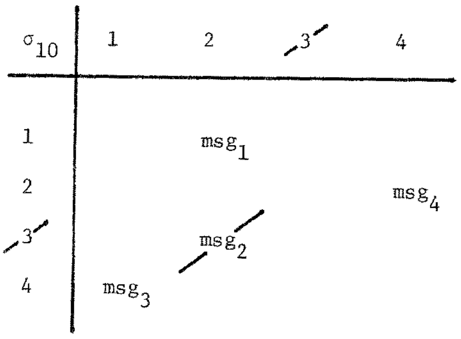

Figure 1

The above sequence of primitives results in the switch of figure 1 where process 3 is no longer active. Also, the message msg_2 has already been received by process 2. Consider the following example of reduction to illustrate the use of the axioms (supposing that no error situation is encountered):

istheremsg(1,2)(σ_{10})

= if x(1,2)(σ_{10}) eq no-msg then false else true — by axiom 7

= if x(1,2)(σ_9) eq no-msg then false else true — by axioms for x

o o o

= if x(1,2)($\text{send}_1(2,\text{msg}_1)(\sigma_2)$) eq no-msg then false else true — by axioms for x

= if msg_1 eq no-msg then false else true — by axioms for x

= true — by boolean axioms

3. A CALCULUS

We are now going to introduce elements of a calculus being developed for reasoning about parallel programs and illustrate its use for verification by addressing common properties found in message oriented programming such as the deadlock problem. (The condition for the occurrence of a deadlock 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 first step in our analysis is to derive from the code of the parallel program synchronization formulae (sf's) which describe the possible sequences of communication primitives for each of the processes that form the program. The synchronization formula is a kind of regular expression with the operations: sequentiality or concatentation (;), deterministic case statement or union (<u>or</u>), and iteration or transitive closure (*). We consider that the concurrent execution of the sf's (or the processes that form the program) is equivalent to an interleaving of these expressions (sf's). In order to represent the interleaving of the sf's (or the sequence of sends and receives performed by the parallel program), we introduce a general shuffle operation. The associated language generated by the shuffle of the sf's is called the unrestricted language L.

Taking into account the communications data type and consequently the semantics of the communication primitives we know that some of the expressions when evaluated may lead to error. For example, axiom 5 states that if a process is blocked in some environment σ (doing a receiving operation), then it cannot perform any other operation before the corresponding sending operation is executed. As a result of this, the language denoting the correct expressions L_c is defined as the difference between the unrestricted language L and the language formed by the incorrect expressions L_e. The expressions that lead to deadlock situations make up a subset of L_e.

We can now restate the deadlock problem. A message-oriented program will be deadlock free if all possible sequences of sending and receiving operations generated by the execution of its code do not lead to expressions in L that contain circular chains of blocked processes.

We have used a general shuffle operation in order to define the unrestricted language L. If the language L_c is empty ($L=L_e$), then all possible sequences of communication primitives lead to unacceptable expressions. If we assume that the sf's do not contain unpaired primitives (where an unpaired primitive is characterized by a sending or receiving operation in an sf with no corresponding operation in the other sf's), then all possible executions of the program lead to deadlock situations. (Note that the existence of unpaired primitives in the sf's indicates that the program has either superfluous use of primitives or that some primitives which should be there have been left out.) On the other hand, in the most common situation the nondeterminism involved in the computation of the processes may introduce an exponential number of final expressions that can make impractical a case by case analysis. In the next paragraph, we start describing a better approach to the problem.

The asynchronous nature of message oriented programming determines the possible occurrence of delay situations such as (i) processes sending messages 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 seqeuence 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 behaviors of a process). The set of communication behaviors (subexpressions separated by the operator or) form the corresponding sf. By grouping the related subexpressions -- i.e. the ones that refer to each other and are activiated 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 interrrelated 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 stituation ($L=\phi$).

Having defined a communications data type in the last section, we present a solution for the producer-consumer problem that is deadlock free in order to illustrate the use of our calculus. (See the appendix for the sample program in an Algol-like notation with the symbols "{" and "}" being used in substitution for the usual pair begin/end.) 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 avid (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.

The first step in the proposed technique for deadlock detection in the communication mechanism is to derive the several synchronization formulas 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, the producer and consumer processes by pd and cs, and the processes that control the resources avpl and avit by pl and it, respectively. The symbol ";" denotes sequentiality of actions (as in programs, so note the order of the statements is the opposite of what one would expect in an expression denoting an environment σ) and "or" that more than one expression can be used for the process.

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:
   $$[\underbrace{(r(pd);\ s(pd,w))^{i-1};\ (r(pd);\ s(pd,go))}_{3a}\ \underline{or}\ \underbrace{(r(cs))}_{3b}]\ \text{for } i \in N$$
4. Proprietor of resource avit:
   $$[\underbrace{(r(cs);\ s(cs,w))^{j-1};\ (r(cs);\ s(cs,go))}_{4a}\ \underline{or}\ \underbrace{(r(pd))}_{4b}]\ \text{for } j \in N$$

First of all we note that there are no unpaired primitives in our sf's. 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 sequences 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)), producer signals one more available item, corresponds to the second subexpression of the fourth expression (r(pd)). Symmetrically, the exponent j used in expression 2 can be related to the corresponding alternatives in expressions 3 and 4.

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 Π 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.

4. CONCLUSIONS

The report presents an outline of a model of communication primitives using the algebraic theory of abstract data types as the specification technique. The model is simple, yet realistic for certain kinds of communicating systems. There is absolutely no reason why the axiomatisation could not be modified to model more complex systems. We further claim that the model is at an appropriate level of abstraction to be useful in the development and specification of message oriented programs, a property which is absent from previous models [6,7].

The elements of a simple calculus based on the specification of the data type has been found to be useful for studying some properties (such as deadlock) of communicating systems. We demonstrated this by showing that a formulation of the consumer and producer problem is deadlock free.

Clearly, directions for further research can be divided into three categories:

(i) Modifications of the axiomatisation to take into account more complex communications. Along these lines, some connection must be made between the communications data type and the semantics of the host language (perhaps also expressed in an algebraic setting).

(ii) Improvements to the calculus to extend its applicability to a larger class of problems. Again, a connection of the calculus with some appropriate calculus for the verification of program properties is desirable.

(iii) Development of program specifciation methods based on the model.

We hope to carry on our work in all three areas of interest.

5. REFERENCES

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

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

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

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

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

[6] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: On the Design and Specification of Message Oriented Programs, to appear in the Int. J. of Computer and Information Sciences.

[7] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: A Methodology for Message Oriented Programming; to be presented at the 6th GI Conference on Programming Languages and Program Development, Darmstadt, March 1980.

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

[9] 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, RC6487, 1976.

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

[11] Hewitt, C., Baker, H.: Laws for Communicating Parallel Processes; Information Processing 1977, pp. 987-992.

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

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

[14] Jammel, A.J., Stiegler, H.G.: Managers versus Monitors; Proc. of the IFIP 1977, pp. 827-830.

[15] Kahn, G., MacQueen, D.B.: Coroutines and Networks of Parallel Processes; Information Processing 1977, pp. 993-998.

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

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

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

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

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

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

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

APPENDIX

```
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}
```