MESSAGE ORIENTED PROGRAMMING - A RESOURCE BASED METHODOLOGY

By

P.R.F. Cunha*, C.J. Lucena+
T.S.E. Maibaum*

Research Report CS-80-32
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada.

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

+Departmento de Informatica
Pontificia Universidade Catolica
Rua Marques de S. Vincente, 225
Gavea - CEP 22453
Rio de Janeiro, RJ, Brazil

This work was supported by a grant from the Natural Sciences and Engineering Research Council of Canada and the Canadian International Development Agency.

ABSTRACT

The development of new technology and hardware systems provides many new opportunities for their exploitation. These opportunities also present us with certain responsibilities - namely, to develop appropriate tools for the orderly and well founded management of these systems. In this report we present a methodology for the development (and analysis) of programs based on the structuring principle of processes which synchronize their activities by message passing. The basis of the methodology is the resource and this concept generalizes that of abstract data type to the message-passing environment. The methodology is a natural outgrowth of methodologies for sequential programs and parallel programs developed for shared address spaces. We illustrate our ideas via solutions to some well known problems such as the bounded buffer problem, the readers and writers problem, the dining philosophers problem and the unreliable medium problem.

Keywords:

Methodology, distributed computing, message oriented programming, resource, abstract data types, asynchronous condition formula, synchronization formula, verification.

1. Introduction

Processes are building blocks for 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 since shared variables can be viewed as a special case of message transmission [30]. Moreover, modern technological developments preclude the use of shared address spaces. Today there is no question about the need for the development of methodologies which, starting from well formulated requirements, guide the progress to a solution which is structured, manageable, and meets these initial requirements [10, 8 , 27, 28].

Modern programming design methods are based on sound principles such as structured programming [10], modular design [9, 26] and programming with abstract data types [21]. These criteria make possible structurable and manageable programs which facilitate the understanding and the verification of the system. There is no question about the need for the development of methodologies where we can state formally a procedure by which a reliable piece of software can be built. The general hope is that starting from well formulated requirements and using the guidelines from the methodology, we should progress to a solution which meets these initial requirements. The methodology should provide helpful criteria for structural partition of the system and for refinement of intermediate solutions.

Motivated by the above considerations, we develop in this paper the beginnings of a methodology to deal with what we call message oriented programming which refers to programming with processes and messages. In terms of problem solving, two interrelated programming problems present themselves: how much of the program should be contained in the message structure and how much in the process structure. The inherent excessive flexibility of message oriented programming suggests that careful attention be paid to the methodological issues that it raises. (This inherent flexibility arises from the possibility of being able to modularise a system in various ways depending on various criteria: Are messages explicit ("send", receive") or implicit (operation invocation [17, 19]? What "size" of message is allowed (i.e. signals or any data item)? What kind of primitives are allowed (i.e. blocked or unblocked sends and/or receives [23])? These are just a few of the many variations which are possible.) We think that the techniques for programming with messages and processes need to be at least as well understood as the techniques for parallel programming with shared variables. Similar concerns are being shown by other researchers in program language design for distributed systems [11, 24] and design of multiprocessor systems [1, 3, 16, 20, 5].

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 ([25, 22, 14]) 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 techniques (more fully described elsewhere [6]) 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 some well known problems: the bounded buffer problem, the readers and writers problem, the dining philosophers problem, the unreliable medium problem.

The first and most important of these design principles is the concept of resource oriented programming. A resource is a simple modularising principle which lends itself very nicely to the modularisation of distributed systems as well to the definition of the internal structure of each module (resource). This principle is derived from various other techniques which have appeared in the literature - namely, programming through managers [18] and proprietors [3]. Another close cousin of the resource concept is the monitor of [15]. We might say that resources are to message oriented programming as monitors are to programming with shared variables. Like monitors, a resource implements an abstract data type [13, 21] together with mechanisms for synchronizing the activities of the resource with its environment (i.e. other resources) as well as internally (since we will allow the use of the resource in parallel by various elements in the environment).

In the next section we outline the programming language which will be the vehicle for the statement of our programs. We also outline a "synchronization language" to be used in program development. We then define and justify the heuristics used in our methodology. In section 3

we develop a number of example programs by using the methodology. Finally, in section 4 we make some remarks concerning the analysis of such programs for the purposes of verification as well as some remarks on the future developments of this work.

2. The Design Methodology

We begin this section with an outline of the languages used in our examples. Then the methodology of message oriented programming is developed.

2.1 The Programming and Synchronization Languages

Our programming language is built around communications primitives (defined in [6]) specified using the algebraic specification technique ([12,13 , 21]). Some of the operations are:

$send_{\ell}(m,msg)$: process ℓ sends message msg to process m;

$send_{\ell}(m)$: a signal from process ℓ to process m. (i.e. the content of the message is unimportant);

$receive_{\ell}(m)$: process ℓ receives a message from process m;

$rec\text{-}any_{\ell}$: returns a pair consisting of process name and message;

$rec\text{-}any_{\ell}(set\text{-}of\text{-}proc)$: process ℓ can receive a message from any process in the given set;

$istheremsg_{\ell}(m)$: a test to determine whether or not there is a message from process m to process ℓ.

Note that the subscripts are dropped in the body of code defining a process as the value of the subscript is obvious in such a situation. Also note that we are using the so-called "unblocked send" and "blocked receive" primitives. This means that a process executing a send immediately goes on to execute the next statement in the program without waiting for the message to actually be received. On the other hand, a receive blocks the program if there is no message to be received until such a message arrives. We chose these particular primitives because they resemble output and input statements in normal sequential programs and will, we believe, lead to simpler analysis of the programs. However, we should say that by using the operation "istheremsg" we can adequately simulate the other possibilities such as blocking send and blocking receive and so on.

The axiomatisation of these primitives is equational and from the algebraic theory of abstract data types we know that we have defined a unique object called the communications data type. Together with assignments and operation invocations, these primitives constitute the basic statements (P) of our programming language (PL). We then define (informally) PL as follows in terms of programs PL1, ..., PLn:

(i) $P \subseteq PL$;

(ii) PL1; PL2 $\subseteq$ PL;

(iii) {PL1} $\subseteq$ PL;

(iv) *{b: PL1} $\subseteq$ PL; (b is a boolean valued expression)

(v) {b1: PL1 or b2: PL2 or ... or bn: PLn} $\subseteq$ PL(b1, ..., bn are boolean valued expressions and $n \geq 1$).

(i) indicates that the basic statements are programs;

(ii) indicates that one program followed by another is a program;

(iii) gives us block structure;

(iv) gives us iteration which we will generally write as "while b do PL";

(v) is Dijkstra's nondeterministic alternative construct if ... fi.

In the special case that we have b: PL1 or ¬b: PL2 we will write "if b then PL1 else PL2". We suppress declarations, etc. in the programs, but we add the reserved word process followed by the corresponding name.

An example of a program in PL is the following (which will appear in the solution to the bounded buffer problem):

```
process p-avpl;
  {avpl := n
   while true do
   {avpl ≠ 0:  msg := receive(prod);
               avpl := avpl - 1;
               place item msg
    or avpl ≠ n:  receive(cons);
                  avpl := avpl + 1;
                  get item into msg;
                  send(cons, msg)}
  }
```

We now define the synchronization language (SL) which will be used in the methodology to develop our programs by specifying the synchronization mechanism for each process. The primitives of the language are s_{ij} to indicate the sending of a message from process i to process j and r_{ij} to indicate the reception of a message by process i from process j. Calling these primitives S and letting SL1,..., SLn be elements of SL we define SL recursively by:

(i) $S \subseteq SL$;

(ii) $SL1; SL2 \subseteq SL$;

(iii) $\{SL1\} \subseteq SL$;

(iv) $*\{SL1\} \subseteq SL$;

(v) $\{SL1 \text{ or } SL2 \text{ or } \ldots \text{ or } SLn\} \subseteq SL$ (for $n \geq 1$).

The interpretation of these various constructs is analogous to those above for PL. The synchronization program in SL corresponding to the example program above is:

```
process p-avpl;
   *{avpl ≠ 0 : receive(prod)
     or avpl ≠ n : receive(cons);
                   send(cons, msg)}
```

We will use the two languages SL and PL, which differ essentially in the amount of detail included in the "programs" of the respective languages in the stepwise development of programs. The first language is used to express synchronization properties of programs and acts as a first approximation to the program in PL which is the final solution.

2.2 The Methodology

Myers [26] 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 relationship 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.

Myers postulates two methods for achieving high strength and low coupling. The first sees a module as implementing a function (in the purely mathematical sense). The second sees a module implementing an abstract data type (i.e. a set of values and operations which can be applied to these values). We have chosen the second alternative as the basis for our methodology.

Informally, the methodology begins with the decomposition of a system into a set of resources which must interact to "solve" the problem at hand. How this interaction takes place is specified by the second major component of a resource-namely, the synchronization part. The first major component is of course the abstract data type which the resource implements. (At the first stage in the stepwise refinement process, we assume that the operations of the data type implemented by the resource can be invoked directly by the programs which manage the resource. Thus the resource's programs are assumed to share a common address space, at

least insofar as they all have direct access to the operations of the resource.) Thus the purpose of the managing programs is essentially the synchronization of these operations. At the next stage in the stepwise refinement process, this assumption of having a shared address space (and thus being able to invoke the operations of the type directly) can be changed to reflect a distributed representation of the data values. This is done by replacing operation invocations by appropriate message passing activities between the appropriate processes and it may involve the use of more primitive resources for the implementation of the resource in question. This stepwise refinement problem is treated in another paper [7].

Granted that we want to do message oriented programming which is based on resource management, the question of how to design our program still remains. In this report we propose a two stage approach: firstly we develop a formula for a given resource, called the asynchronous conditions formula (acf), to define in an explicit way the process structure associated with the management of the resource. This formula will be of the form

$$(c_{11}\wedge\ldots\wedge c_{1n_1}) \underline{\text{ or }} (c_{21}\wedge\ldots\wedge c_{2n_2}) \underline{\text{ or }} \ldots \underline{\text{ or }} (c_{m1}\wedge\ldots\wedge c_{mn_m}) \equiv s_1 \underline{\text{ or }} \ldots \underline{\text{ or }} s_m$$

for some (propositional) truth values c_{ij} $(1\leq i\leq m,\ 1\leq j\leq n_i, n_i\geq 1)$. The motivation for the structure of this formula is the following. Associated with each resource is a set of operations $op_1,\ldots,op_m$ which we would like to use to manipulate the resource. Each of these operations has associated

with it a precondition which must be satisfied before the operation can be applied. Thus s_i is the precondition which must be satisfied before op_i is applied to the resource. This formula defines all conditions (and the only conditions) under which any action concerning the resource can take place. Thus, if $(c_{i1} \wedge \ldots \wedge c_{in})$ is true, then a particular use of the resource may be made. (Note that the truth values c_{ij} may in turn be defined in terms of some acf.) We note here that the conditions which make s_i and s_j true, for $i \neq j$, need not be mutually exclusive. This corresponds to the idea that several parallel operations on a resource may be compatible.

We then have two (opposite) criteria for defining process structures associated with such an acf. The so-called functional strength approach dictates that a single process be associated with the management of the resource. This process has to handle the different conditions $s_1, \ldots, s_m$ and take the appropriate actions. There are a number of possible ways to implement this. The obvious one is to handle $s_1, \ldots, s_m$ by cases (closely related to the use of guarded commands). Another possibility is to use a hierarchical decomposition of this one process by defining processes which handle each (or some subset) of the conditions controlled by some "master" process. The other extreme is informational strength and the criterion used in this case is the explicit handling of the resource considered as a data structure, in terms of operations defined on it. Again the acf is conducive to this kind of structuring since the conditions s_i establish criteria for particular actions to take place in

the management of the resource. Thus the so-called informational strength approach associates with a resource one process for each of the conditions $s_i, 1 \leq i \leq m$. These processes are dedicated in the sense that they manage only a particular aspect of the given resource. This then leads to a highly distributed or "horizontal" organizational structure for these processes because we are looking for operations that can be executed in parallel.

Having established the process structure for the problem at hand, can we now find some guide to help us design the processes themselves. The following analysis leads us to a solution. Let us consider the interaction of a resource with the "outside world" or its "environment". This interaction is accomplished purley via message passing. Messages are received from other resources asking for the activation of some operation. Messages are sent to other resources (or internally among processes in the same resource) acknowledging the completion of some operation or passing on results or ... . Thus the sequence of message passing actions in the system is of vital importance in defining the internal structure of processes associated with the resource in question. This suggests again a solution based on a formula - the synchronization formula (sf) $f_1 f_2 \ldots f_n$, where each f_i is some primitive communcations activity like send or receive and each f_i identifies the process(es) involved in the communications action. This formula concentrates on the communications activities of a given process and specifies the intended synchronization of the process with its environment via its message passing activities. This formula is specified as a program written in the synchronization language SL.

Having established the sf, the message passing and receiving activities of each process are now well defined and the appropriate actions to be taken on the occurrence of each such communication have to be "filled in". Normal structured programming techniques can be used to do this since these parts of the processes are implemented by "normal" (non-communicating) algorithms.

The methodology can then be summarized as follows:

(i) Identify the resources needed to solve the problem at hand. The concept of data type is the main guide in this identification in the sense that we want to group together in one resource what is normally implemented as a cluster or a data abstraction in sequential programs.

(ii) Having identified the resources, assume that the data type which the resource manages has been implemented and thus the operations associated with the resource are known as are the properties of these operations.

   a) Establish for each resource the asynchronous conditions under which various operations may be invoked.

   b) Establish the process structure for each resource by using the criteria of functional strength or informational strength (or some compromise between the two).

(iii) Establish the synchronization formula (sf) for each process involved in the management of a given resource.

(iv) "Fill in" the sequential parts of each process. That is derive a program in the language PL by using the program in SL as a "skeleton".

Our intention in the next section is to illustrate these ideas via the solutions to a number of interesting problems.

3. Examples

This section contains solutions to familiar problems of parallel programming using the methodology developed previously. For some examples we give more than one solution using the possible criteria involved in the technique. (Moreover, different levels of refinement are presented for the same solution.) The comparison of the functional and informational strength approaches helps us to understand the differences between the implementation of the resource in a shared address space and a distributed one. We start by presenting in subsection 3.1 a detailed illustration of our methodology by giving diverse solutions to the bounded buffer problem. In subsection 3.2, other well known problems are solved without the provision of too many details.

3.1 A Detailed Example

To illustrate the method described above we will develop a detailed message-oriented programming solution to the bounded buffer problem. This problem can be stated in the following way: a bounded buffer is constructed in order to smooth variations in the speed of output by a producer process and input by a consumer process [16]. The producer and the consumer processes repeat their actions continuously and it is known that the buffer area is large enough to hold n items.

Taking the steps presented in the methodology, we have:

(i) Identification of the resources:

In this case, we are interested in the synchronization between the buffer area and the producer and consumer processes. The buffer area can be thought of as a data type where the items are the objects and the operations are "place an item" and "get an item". For this example, we consider the producer and consumer as external resources which interact with the buffer-area resource. The resource directly involved in the problem is the buffer area and its length of n buffers determines how many messages it can store at any time.

(ii) Establishment of the acf:

The resource identified for this problem is the buffer area. We assume that the data type (and consequently the operations) associated with the resource has been implemented. In this case, the operations are "place an item at last" and "get an item from first" in which "first" and "last" are pointers to, respectively, the next available item and the next available slot in the buffer area. Let us consider, for the purpose of defining the asynchronous condition formula (acf) associated with the resource, the following two predicates:

- NE - buffer area is not empty
- NF - buffer area is not full

(If we consider the variable k as the current number of messages in the buffer area, it follows that NE $\equiv k > 0$ and NF $\equiv k < n$.)

The consumer process must not try to get a message from the buffer area if this is empty. Similarly, the producer process must wait if the buffer area is full. In view of this, the conditions under which the operations "place an item" and "get an item" may be invoked are "NF" and "NE". Therefore, the asynchronous condition formula (acf) for the bounded buffer problem is expressed as follows:

NF or NE ≡ acf - producer or acf - consumer

Firstly, we consider a solution based on the criterion of functional strength. As defined previously, this approach associates one process for the management of the resource. A single process will handle by cases the conditions s_1 and s_2 (acf - producer and acf - consumer) using the alternative construct. We are not designing the producer and consumer process because our main interest here is the control of the buffer area. We assume the usual sequence of communications primitives for a producer or a consumer. The producer contains the primitive send(avpl, item) and the consumer the pair of primitives (send(avit) ; x := receive(avit)).

(iiia) Establishment of the sf:

The synchronization formula (sf) is a representation of the intended sequence of all communication operations ("sends" and "receives") performed by a process. The set of synchronization formulas (one for each process) specifies the intended synchronization of the system via its message passing activities.

In the functional strength approach, one process treats the producer and consumer cases. The operations "place an item" and "get an item" are invoked in the same process that manages the resource (buffer area) and consequently, they are mutually exclusive. (Remember that we are assuming here a shared address space for the resource's program.) The synchronization formula for the process p-avpl (number of available places in the buffer area) used in the functional strength design is given below. This expression (sf) gives the necessary sequence of communication primitives for the process p-avpl in order to handle the instances of messages from the producer and consumer processes as described previously. In the process p-avpl we have the producer case and the consumer case subdivided by the respective conditions "avpl ≠ 0" and "avpl ≠ n". Let us denote receive by r, send by s, producer by pd and consumer by cs in the following expression. The symbol ";" denotes sequentiality of actions, "*" an indefinite number of repetitions of the enclosed communication sequence and "or" that the expressions are disjoint in the code used for the process.

1. $[\underbrace{(r(cs)\ ;\ s(cs,\ item))}_{avpl \neq n\ (\text{"cs" case})}\ \text{or}\ \underbrace{r(pd)}_{avpl \neq 0\ (\text{"pd" case})}]^*$

   (The expression inside the square brackets is intended to specify one instance of the communication activity of the process p-avpl with the producer or consumer.The same history will then be repeated for the next activation of the producer or consumer.)

Using the sf established above, it is easy to derive the communication skeleton (or the synchronization program expressed in SL) of the process p-avpl. We know that avpl ≠ 0 (NF) is the condition under which the operation "place an item" can be invoked and avpl ≠ n (NE) is the corresponding condition for the operation "get an item". The mapping is almost direct considering ";" as the usual delimiter of statements, "*" as iteration and "or" as an indication of mutually exclusive sequences of communication primitives. The program in the synchronization language (SL) is given below as a first approximation of the final program in PL.

```
process p-avpl ;
  * { avpl ≠ 0 : receive(prod)
    or avpl ≠ n : receive(cons);
                  send(cons, msg)}
```

(iv.a) Filling in of the sequential part:

The filling in of the sequential part is the last step. This part is independent of the communication mechanism and it may depend on implementation details. One possible final form for the process p-avpl is given below. We use the previous program in SL as a skeleton to derive the final program in the language PL.

```
process p-avpl ;
  { avpl ≠ 0 : msg := receive(prod) ;
               avpl := avpl -1 ;
               place item msg
  or avpl ≠ n : receive(cons) ;
               avpl := avpl + 1 ;
               get item into msg ;
               send(cons, msg)     }
}
```

Secondly, we consider the criterion of informational strength. This approach associates with the resource one process for each of the pre-conditions s_1 and s_2 (where s_1 = acf - producer and s_2 = acf - consumer). As indicated before, each process will manage a particular aspect of the resource. It is possible to base the solution of the problem on the processes p-avpl (number of available places (avpl) greater than zero) and p-avit (number of available items (avit) greater than zero). The first process will treat the operation "place an item" from the producer if "avpl ≠ 0" (assuring NF) and the latter will treat the operation "get an item" to the consumer if "avit ≠ 0" (assuring NE).

(iii.b) Establishment of the sf:

Based on the discussion above we have to design two process p-avpl and p-avit to handle the operations "place an item" and "get an item", respectively. The process p-avpl can receive messages from the producer or the companion process p-avit. In the first case, the producer sends

an item to be put into the buffer area while in the second case, the process p-avit informs p-avpl that a new slot is available in the buffer area. If "avpl = 0", the process p-avpl which cannot receive messages from the prcducer blocks itself waiting for a signal from the process p-avit; otherwise, it is free to receive messages from the producer or p-avit. When the process p-avpl places an item in the buffer, it signals the process p-avit that a new item is available. The companion process p-avit, related to the consumer case, has a behaviour similar to that of the process p-avpl when considering the dual condition "avit = 0".

This informational strength design is less restrictive than the functional strength one. The operations "place an item" and "get an item" can be executed concurrently in this informational strength design and therefore more parallelism is achieved. It is only required to assure that these operations do not access concurrently the same buffer slot. This condition is ensured because the process p-avpl only informs the process p-avit that a new item is available after it has finished the work related to that item. The same happens in the inverse case when a new place is available. (Of course, there is not much advantage if this solution is implemented on a single processor.)

The sf expressions for the two process p-avpl and p-avit used in the informational strength design are given below. These expressions give the necessary sequence of communication primitives between the process p-avpl, p-avit, producer and consumer as explained before. For example, the process p-avpl communicates with the process producer and p-avit, and it handles the conditions "avpl = 0" and "avpl > 0".

1. Process p-avpl:

$$[\underbrace{(r(pd)\ ;\ s(p\text{-}avit)\ \text{or}\ r(p\text{-}avit))}_{avpl > 0}\ \text{or}\ \underbrace{r(p\text{-}avit)}_{avpl = 0}]^*$$

2. Process p-avit:

$$[\underbrace{(r(cs)\ ;\ s(cs,\ item)\ ;\ s(p\text{-}avpl)\ \text{or}\ r(p\text{-}avpl))}_{avit > 0}\ \text{or}\ \underbrace{r(p\text{-}avpl)}_{avit = 0}]^*$$

(When writing an sf, we use the other written expressions (sf's) in order to help the design of this sf or to validate the whole message passing activity. Comparing the expressions 1 and 2 above, we can verify easily that there is a complete match of the sending and receiving operations between them. The complete match is also true, if we relate these expressions with the communication primitives in the producer and consumer processes. A set of sf's with a complete match of sending and receiving operations is said to have no unpaired primitives.)

Using the sf's established previously, it is easy to derive the synchronization program expressed in SL for the processes p-avpl and p-avit. For example, in the process p-avpl where the condition "avpl = 0" is true, it blocks itself until a place becomes available; otherwise, it can receive messages from both the processes producer and p-avit.

```
process p-avpl
  *{ avpl > 0 :
        { receive(prod) ;
          send(p-avit)
        or receive(p-avit) }
   or avpl = 0 : receive(p-avit) }

process p-avit
  *{ avit > 0 :
        { receive(cons) ;
          send(cons, msg) ;
          send(p-avpl)
        or receive(p-avpl) }
   or avit = 0 : receive(p-avpl) }
```

(iv.b) Filling in of the sequential part:

We use the programs in SL as skeletons to write the final programs in PL. One possible version of the processes p-avpl and p-avit in the informational strength design is given below.

```
process p-avpl
  { avpl := n ;
    while true do
    { avpl > 0 : t := rec-any ;
          { t.name = 'prod' : avpl := avpl - 1 ;
                              place item t.msg ;
                              send(p-avit)
          or t.name = 'p-avit' : avpl := avpl + 1 }
    or avpl = 0 : t := receive(p-avit) ;
                  avpl := avpl + 1     }
  }
process p-avit;
  { avit := 0 ;
    while true do
    { avit > 0 : v := rec-any ;
          {v.name = 'cons' : avit := avit - 1 ;
                             get item in v.msg ;
                             send(cons, v.msg) ;
                             send(p-avpl)
          or v.name = 'p-avpl' : avit := avit + 1 }
    or avit = 0 : v := receive(p-avpl);
                  avit := avit + 1    }
  }
```

(As mentioned in the last section, here we are assuming that the processes p-avpl and p-avit use a shared address space and thus are able to invoke the operations "place an item" and "get an item" of the data type "buffer" directly. The stepwise refinement process where we consider a distributed representation of the data values is described in [7]. Note also that with a small change in this solution is applicable if there is more than one producer or consumer process.)

3.2 Other Examples

This subsection contains other examples of the use of our resource based methodology to derive message oriented programs. Some of the details of the design process will not be repeated here and we will concentrate mainly on the characteristics of the new examples.

a) The Unreliable Medium

We want to design a program to model the unreliable medium in a communication system. The medium receives a message from a process in the external environment and takes nondeterministically one of the following actions: (i) sends the message intact to the receiving process, (ii) delivers the message with an error signal, or (iii) skips that message (i.e., it looses the message). This resource is part of the specification of the "alternating bit" protocol described in Bochmann et al. [2]. Taking the steps presented in the methodology, we have the solution below.

i) Identification of the resources:

The resource to be designed here is the unreliable medium. This resource can be viewed as a data type where the messages in transit are the objects and the operations are scorr (send a correct copy of the message), sinc (send an incorrect copy of the message) and skip (skip the present message).

ii) Establishment of the acf:

The asynchronous condition formula (acf) for the resource "unreliable medium", which expresses the conditions under which the operations of the data type may be invoked, is now developed.

ii) From the definition of the problem, we know that the operations scorr, sinc, and skip should be mutually exclusive (the medium arbitrarily just performs one of these operations). Thus, the acf for the resource above is expressed as follows:

(NIS ∧ NS) or (NCS ∧ NS) or (NCS ∧ NIS) ≡ acf-scorr or acf-sinc or acf-skip

where NCS, NIS, NS are the conditions "no correct sending", "no incorrect sending", and "no skipping", respectively.

iii) Establishment of the sf:

A functional strength solution will be designed for the resource "unreliable medium" since it is not possible to achieve a higher degree of parallelism (or any other apparent advantage) by using more than one process to manage the resource. The following synchronization formula (sf), which is trivial in this case, establishes the sequence of message passing primitives that the process medium will use to synchronize its activities with the other processes (senders and receivers).

[r(sender-i); (s(receiver-k, msg) or s(receiver-k, 'error') or nil)]*

where sender i addresses receiver k.

iii) The program in the synchronization language (SL) which represents a first approximation of the final program is given below:

```
process medium ;
  *{ receive(sender-i) ;
     { send(receiver-k, msg)
       or send(receiver-k, 'error')
       or skip                    }}
```

iv) Filling in of the sequential part:

The acf given above is guaranteed in the solution because the three conditions (separated by or in the acf) are handled by the same process and therefore the operations are executed sequentially (each time the medium performs one of the operations). In this case there is no sequential part to be filled in the previous program. Rewriting the above code in the programming language PL, we have the following program for the resource medium.

```
process medium ;
 { while true do
     { t := rec-any ;
       { send(t.name, t.msg)
         or send(t.name, 'error')
         or skip                }}
```

b) The Readers and Writers

The problem of the readers and writers was originally proposed by Courtois et al. [4]. The problem can be stated in the following way. Readers and writers are processes which share a resource. The readers can use the resource (e.g., a data base) 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 (this is the so called writer priority version). Below a solution is developed using our resource based methodology.

i) Identification of the resources:

The design of this problem involves the resources readers-writers and scheduler. In the first resource we assume that the operations read and write of the data type "data area" have been implemented and they can be invoked directly by readers and writers. The scheduler controls the sequence of accesses by readers and writers. We are not going to design completely the readers and writers processes because our main interest here is the resource scheduler. The operations of the scheduler are gread (grant reading condition) and gwrite (grant writing condition). A reader process asks permission of the scheduler before it reads some data and when finished it signals the conclusion of the operation. (The situation is analogous for a writer process.) The parts of a reader process and a writer process that refer to the scheduler are given below. (In general, a reader or a writer may refer to other resources.)

```
process reader-i ;
  { . . .
     send(sched, 'sread') ; receive(sched) ;
     read(data-area) ;
     send(sched, 'eread') ;   . . .      }
process writer-i ;
  { . . .
     send(sched, 'swrite') ; receive(sched) ;
     write(data-area) ;
     send(sched, 'ewrite') ; . . .       }
```

ii) Establishment of the acf:

The resource scheduler will control the sequence of reading and writing in order to assure that these two operations do not overlap and also a higher priority for writing. The conditions under which the reading and the writing operations can be granted are NWA $\wedge$ NWR (no reading activity and no writing request) and NWA $\wedge$ NRA (no writing activity and no reading activity), respectively. Therefore, the acf for the scheduler is:

(NWA $\wedge$ NWR) or (NWA $\wedge$ NRA) $\equiv$ acf-reading or acf-writing

iii) Establishment of the sf:

As we know, in the functional strength approach we associate with the resource one process which treats the conditions of the acf by cases. The solution for this approach has been extensively published in the literature. In view of this, we are going to design an informational strength solution for the readers and

iii) writers problem. This solution associates with the resource one process for each of the conditions acf-reading and acf-writing. We will have two processes: r-sched (reading scheduler) and w-sched (writing scheduler), to grant reading and writing permissions, respectively. Although the operations read and write cannot be activated at the same time, some parallelism is gained because now process r-sched will treat the messages "sread" and "eread", and process w-sched the messages "swrite" and "ewrite". (The solution is also a nice example of cooperation between these two processes.) The design of the message structure of the process r-sched proceeds as follows: if there is a message from process w-sched signalling a writing request then process r-sched waits until all the readings that are in progress end to signal the writing scheduler to go ahead. After the writer finishes the reading scheduler is unblocked. (The design of the message structure of the process w-sched matches the primitives above and it is straightforward.) The sf's for both processes r-sched and w-sched are written below:

1. Process r-sched:

[(r(w-sched) ; (r(readers-in-progress))*; s(w-sched, 'go');
r(w-sched)) or
(r(reader-i) ; (s(reader-i, 'oktoread') or nil)]*

2. Process w-sched:

[r(writer-i); s(r-sched) ; r(r-sched); s(writer-i, 'oktowrite');
r(writer-i); s(r-sched, 'go')]*

Using these sf's, it is easy to derive the following program in SL for the processes r-sched and w-sched. The condition istheremsg(w-sched) assures the writing priority, the variable msg stores the message received, and the variable rcount keeps track of the number of readings that are still in progress.

```
process r-sched;
  *{istheremsg(w-sched) :
       {receive(w-sched) ;
          *{rcount ≠ 0 : receive(rprogress)}
          send(w-sched) ;
          receive(w-sched) }
   or listheremsg(w-sched) :
       {receive(reader-i) ;
         {msg = 'sread' : send(reader-i, 'oktoread')
          or msg = 'eread' : nil                    }}}
```

```
process w-sched ;
  *{ receive(writer-i) ;
     send(r-sched) ;
     receiver(r-sched) ;
     send(writer-i, 'oktowrite') ;
     receive(writer-i) ;
     send(r-sched)     }
```

iv) Filling in of the sequential part:

We give below the code (in the language PL) for the processes r-sched and w-sched which form the resource scheduler. We make use of the data type "set" for which the operations insert and delete are defined in the usual way (sreaders and swriters denote the sets of readers and writers, respectively).

```
process r-sched ;
  { rcount := 0 ;
    while  true  do
       if istheremsg(w-sched)
       then { receive(w-sched) ;
              while rcount ≠ 0 do
                 { t := rec-any(rprogress) ;
                   rcount := rcount -1 ;
                   delete(rprogress, t.name)}
              send(w-sched) ;
              receive(w-sched)       }
       else { t := rec-any(sreaders) ;
              { t.name = 'sread' :
                  send(t.name, 'oktoread') ;
                  rcount := rcount + 1 ;
                  insert(rprogress, t.name)
              or  t.name = 'eread' :
                    rcount := rcount - 1 ;
                    delete(rprogress, t.name) }}
  }
```

```
process w-sched ;
  { while  true  do
      { v := rec-any(swriters) ;
        send(r-sched) ;
        receive(r-sched) ;
        send(v.name, 'oktowrite') ;
        receive(v.name) ;
        send(r-sched)       }
  }
```

c) The Dining Philosophers (E.W. Dijkstra)

Five philosophers spend their lives thinking and eating. When a philosopher is hungry, he enters the dining room and sits in his own chair at a circular table which is set with five plates of spaghetti and five forks (one between each plate). Because the spaghetti is so tangled, a philosopher has to use both forks (the ones on the left and on the right of his plate) to eat. When a philosopher has finished eating, he puts down both forks and leaves the dining room.

i) Identification of the resources:

This problem can be solved by considering six resources: The five philosophers and the dining room. As we know, the design of the resource philosopher (identical for each of the philosophers) is quite simple and involves the repetition of the sequence "THINK ; EAT". Before starting the operation of eating,

a philosopher signals his itentions to the resource room which is responsible for access to the forks, and after he is finished eating, he signals the conclusion of the operation. For another application of the ideas presented in our resource based methodology, we will concentrate on the design of the resource called room. The code of the ith resource phil-i may be described as follows:

```
process phil-i ;
  { THINK ;
    getforks ;
    EAT ;
    releaseforks }
```

ii) Establishment of the acf:

The resource to be developed is the dining room. This resource controls the access to the forks by keeping a list of the philosophers that are currently eating. We assume the implementation of the two data types "forks" and "eating-i" (an array of Boolean variables in which eating(i) is equal to "true" if philosopher "i" is eating and "false" otherwise) for the design of the resource dining room. The operations for the second data type are valeat (gets the value of the variable eating (i)) and wrteat (writes the values true (or 1) or false (or 0) into the variable eating (i)). (Note that these two operations can be executed in parallel and no special condition is necessary to hold for their invocation. This is because the entries in the

array can assume only the values true (1) or false (0). See the note after the presentation of the solution for further clarification.) The operations for the data type forks are getforks(phil(i)) (get forks for philosopher i) and release-forks(phil(i)) (release forks from philosopher i), and the conditions under which these operations can be invoked are expressed in the acf below. For the purpose of defining the acf, consider the following two predicates : NE(i) (philosopher i is not eating) and E(i) (philosopher i is eating). Then, we have the following acf:

$(NE(i \ominus 1) \wedge NE(i \oplus 1))$ or $(E(i)) \equiv$ acf-gforks or acf-rforks

where $\oplus$ and $\ominus$ denote addition and subtraction modulo 5, respectively.

iii) Establishment of the sf:

Using the informational strength approach we base the design of the resource dining room on the processes gforks (one which controls the access to the forks) and rforks (one which controls the release of the forks). The first process waits until the condition acf-gforks is true for some philosopher in order to receive his request for eating, allocate the corresponding forks and signal the philosopher back that he can start eating. The latter process, as soon as the philosopher sends a signal stating that he finished eating, releases the forks. The sf's

(which are trivial in this case) for the process phil-i (ith philosopher), the process gforks and the process rforks are written below:

1. Process phil-i (all philosophers have a similar behaviour):

   [s(gforks) ; r(gforks) ; s(rforks)]*

2. Process gforks:

   [r(phil-i) ; s(phil-i)]*

3. Process rforks:

   [r(phil-i)]*

The program in the synchronization language (SL) which represents a first approximation of the final program is given as follows:

```
process phil(i:0 .. 4) ;
  *{ send(gforks) ;
     receive(gforks) ;
     send(rforks) }
process gforks;
  *{(i : 0 ... 4) (istheremsg(phil(i)) ∧
                   ⌉ valeat(i ⊖ 1) ∧  ⌉ valeat(i ⊕ 1)) :
                   receive(phil(i)) ;
                   send(phil(i))       }
process rforks;
  *{(i : 0 .. 4) (istheremsg(phil(i)) ∧ valeat(i)) :
                   receive(phil(i))      }
```

(The condition istheremsg(phil(i)) assures that philosopher i is ready to eat. Note also that the range (i : 0 .. 4) given in the guarded commands above is simply a short form for a nondeterministic construct with five guarded commands - one for each of the permitted values of the variable i.)

iv) Filling in of the sequential part:

The programs in SL are used as skeletons to write the final form of the processes defined previously. We write the header "process phil(i : 0 .. 4)" to express the fact that all the philosophers have similar code.

```
process phil(i : 0 .. 4) ;
 { while true do
     { THINK ;
       send(gforks) ;
       receive(gforks) ;
       EAT ;
       send(rforks)  }
 }
```

```
process gforks ;
  { while true do
      { (i : 0 .. 4) (istheremsg(phil(i)) ∧
                      ⌉ valeat(i ⊖ 1) ∧ ⌉ valeat(i ⊕ 1)) :
                      receive(phil(i)) ;
                      getforks(phil(i)) ;
                      send(phil(i)) ;
                      wrteat(i, 1)          }
      }
process rforks ;
  { while true do
      {(i : 0 .. 4) (istheremsg(phil(i)) ∧ valeat(i)) :
                      receive(phil(i)) ;
                      releaseforks(phil(i)) ;
                      wrteat(i, 0)          }
  }
```

Note that the operations valeat in process gforks and wrteat in rforks could conflict in principle. However, if rforks sets eating(i ⊖ 1) (or eating(i ⊕ 1)) to false (0) then one of two things may happen. Either valeat(i ⊖ 1) (or valeat(i ⊕ 1)) is true and so the guard in gforks is false and philosopher i is delayed or valeat(i ⊖ 1) (or valeat(i ⊕ 1)) is false and so, depending on the values of the other expressions in the guard in gforks, philosopher i may get the forks and begin eating. The worst that can happen is that philosopher i is delayed in his attempt to eat.

The informational strength design here is much less restrictive than the functional strength one because the operations getforks and releaseforks can work concurrently. The resource based methodology conducted us to a concise, elegant and modular solution which also contains a high degree of parallelism. We are going to present below the functional strength solution (in PL) for the dining philosophers problems in order to allow the readers a chance to compare both solutions.

```
process phil(i : 0 .. 4) ;
  { while  true  do
      { THINK ;
        send(room,'enter') ;
        receive(room) ;
        EAT ;
        send(room, 'exit') }
  }

process room ;
  { while  do  true
      {(i : 0 .. 4) (istheremsg(phil(i)) ∧
                  ¬ valeat(i ⊖ 1) ∧ ¬ valeat(i ⊕ 1)) :
                  msg := receive(phil(i)) ;
                 {msg = 'enter' : getforks(phil(i)) :
                                  send(phil(i)) ;
                                  wrteat(i, 1)
                 or msg = 'exit' : releaseforks(phil(i)) ;
                                   wrteat(i, 0)        }
      }
  }
```

4. Concluding Remarks

The development of new technology and hardware systems provides many new opportunities for their exploitation. These opportunities also present us with certain responsibilities - namely, to develop appropriate tools for the orderly and well founded management of these systems. We have tried in this report to present a methodology for the design of programs based on the structuring principle of processes which synchronize their activities by message passing.

The basis of the methodology is the resource. This concept generalizes that of abstract data type to the message passing environment. The purpose of a resource is to co-ordinate the use of the operations of the data type with each other and with the environment of the resource (i.e., the "users" of the operations).

The methodology consists of a sequence of steps of which the following is a summary. The first step is the identification of the resources which will be used in the design of the system being developed. This is analogous to the identification of the abstract data types which would be used in the design of a sequential program. The second step is the establishment of the asynchronous condition formula (acf) for each resource. This formula determines the degree of parallelism which is allowed in the use of the operations of the resource. The third step uses the acf to establish a process structure for the resource by using the criteria

of functional strength, informational strength, or some compromise between them. Once a particular process structure has been chosen, a synchronization formula (sf) must be defined for each process. This formula establishes (as a regular expression) the sequence of sends and receives which the process will perform to co-ordinate its activities with other processes in the resource and the resource's environment (i.e., invokers of the operation(s) controlled by the process). At this point, it is assumed that the operations of the resource can be invoked directly. Thus the assumption amounts to having a correct implementation of the operations on an address space shared by all the processes managing the operations of the resource. (An important point not considered in this report is how this correct implementation can be achieved and how distributed implementations are constructed. This is dealt with in another report [7].) The final step is the "filling in" of the program from the sf defining the structure of the process.

The methodology is a natural outgrowth of methodologies for sequential programs and parallel programs developed for shared address spaces. The question remains as to how properties of such programs may be verified. We do not intend to give a full treatment here but give a brief description of the concepts. The two main criteria for checking the correctness of an implementation of a resource are the verification of the validity of the acf and the confirmation of certain synchronization properties (such as the absence of deadlock). The former involves two steps. The first of these is the proof that the acf

correctly describes the conditions under which operations may be invoked. This amounts to showing that the part of the acf pertaining to a particular operation is an invariant for the process managing the operation at the point(s) where the operation is invoked. The second part consists of showing that the properties of the operations of the resource are not violated by the possible parallel invocation of the operations, assuming that operations are invoked only under the conditions specified by the acf.

Synchronization properties are such things as absence/presence of (partial) (potential) deadlock, starvation, unpaired primitives (a send or receive in a process for which there exists no corresponding receive or send in the corresponding process) use of unbounded buffers, etc. That these properties can be treated in a systematic way is demonstrated in [6,29].

We expect to refine the methodology further in the future and work towards a complete system for the development and verification of message oriented programs based on the concept of resource.

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. BOCHMANN, G.V., GECSEI, J.: A Unified Method for the Specification and Verification of Protocols; Proceedings of the IFIP77.

3. CHERITON, D.R., MALCOLM, M.A., MELEN, L.S., SAGER, G.R.: Thoth, A Portable Real-Time Operating System; CACM, February 1979.

4. COURTOIS, P.J., HEYMANS, F., PARNAS, D.L.: Concurrent Control with "Readers" and "Writers"; CACM, October 1971, (pp. 667-668).

5. CUNHA, P.R.F., LUCENA, C.J., MAIBAUM, T.S.E.: On the Design and Specification of Message Oriented Programs; Research Report CS-79-25, University of Waterloo, June 1979 (to appear in the Int. J. of Computer and Information Sciences).

6. CUNHA, P.R.F., MAIBAUM, T.S.E.: A Communications Data Type for Message Oriented Programming; Lecture Notes in Computer Science, Springer-Verlag, Vol. 83, 1980.

7. CUNHA, P.R.F., MAIBAUM, T.S.E.: "Resource = Abstract Data Type + Synchronization" - A Methodology for Message Oriented Programming; Research Report CS-80-28, University of Waterloo, May, 1980.

8. DAHL, O.J., HOARE, C.A.R.: Hierarchical Program Structures; Structured Programming, Academic Press, London, 1972.

9. DENNIS, J.B.: Modularity; an Advanced Course on Software Engineering, Ed. F. Bauer, Springer-Verlag, 1973.

10. DIJKSTRA, E.W.: Notes on Structured Programming; Structured Programming, Academic Press, London, 1972.

11. FELDMAN, J.: High Level Programming for Distributed Computing, CACM, Vol. 22, June 1979, (pp. 353-368).

12. 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.

13. GUTTAG, J.V.: Abstract Data Types and the Development of Data Structures; CACM, Vol. 20, No. 6, 1977, (pp. 396-404).

REFERENCES - Cont'd.

14. HEWITT, C., BAKER, H.: Laws for Communicating Parallel Processes; Information Processing 1977, pp. 987-992.

15. HOARE, C.A.R.: Monitors, an Operating System Structuring Concept; CACM, October, 1974 (pp. 549, 557).

16. HOARE, C.A.R.: Communicating Sequential Processes: CACM August, 1978 (pp. 666-677).

17. ICHBIAH, J.D., et al.: Preliminary ADA Reference Manual; Sigplan Notices, Vol. 14, No. 6, June 1979.

18. JAMMEL, A.J., STIEGLER, H.G.: Managers Versus Monitors; Proceedings of the IFIP 1977 (pp. 827-830).

19. LAUER, H.C., NEEDHAM, R.M.: On the Duality of Operating Systems Structures; Proceedings of the 2nd International Symposium on Operating Systems, IRIA, Oct. 1978.

20. LAUER, P.E., TORRIGIANI, P.R., SHIELDS, M.W.: COSY - A System Specification Language Based on Paths and Processes; Acta Informatica 12, Springer-Verlag, 1979, (pp. 109-158).

21. LISKOV, B.H., ZILLES, S.: Programming with Abstract Data Types; Proc. Conference on Very High Level Lanugages, SIGPLAN, Vol. 9, April 1974.

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

23. MANNING, E., LIVESEY, N.J., TOKUDA, H.: Interprocess Communication in Distributed Systems - One View; to appear in the Proceedings of the IFIP 80, North Holland, Oct. 1980.

24. MAO, T.W., YEH, R.T.: Communication Port - A Language Concept for Concurrent Programming; IEEE Trans. on Soft. Eng., SE-6, 2(March 1980), (pp. 194-204).

25. MILNE, G., MILNER, R.: Concurrent Processes and their Syntax. JACM, Vol. 26, No. 2, April 1979.

26. MYERS, G.J.: Composite/Structured Design; van Nostrand Reinhold Co., 1978.

REFERENCES - Cont'd.

27. PARNAS, D.J.: A Technique for Software Module Specification with Examples; CACM, May 1972, (pp. 330-336).

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

29. ZAFIROPULO, P., WEST, C.H., RUDIN, H., COWAN, D.D., BRAND, D.: Toward Analyzing and Synthesizing Protocols; IEEE Transactions on Communications, April 1980.

30. ZAVE, P.: On the Formal Definition of Processes; Conf. on Parallel Processing, Wayne State University, IEEE Computer Society, 1976.