RESOURCE = ABSTRACT DATA TYPE + SYNCHRONIZATION - A METHODOLOGY FOR MESSAGE ORIENTED PROGRAMMING -

by

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

Research Report CS-80-28

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

May, 1980

RESOURCE = ABSTRACT DATA TYPE + SYNCHRONIZATION

- A METHODOLOGY FOR MESSAGE ORIENTED PROGRAMMING -

P.R.F. Cunha

T.S.E. Maibaum

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada
N2L 3G1

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

ABSTRACT

With the enormous growth in the application and use of distributed computing, it is of paramount importance that useful strategies and techniques be developed for the design and implementation of programs suitable for such environments. We present in this paper a methodology for the development (and analysis) of programs designed specifically for distributed environments where synchronization is achieved through message passing. The methodology is based on techniques and concepts which have been found to be useful for the development of sequential programs -- namely, stepwise refinement and abstract data types. The methodology is based on the concept of resource, generalizing the concepts of monitors, managers, proprietors, etc. We put forward the proposition that a resource is an abstract data type together with mechanisms for synchronization: firstly, for the operations of the type with each other (to gain parallelism) and, secondly, to enable the user environment to perform operation invocation. A methodology is then presented for the design of resources and their implementation.

Keywords: Methodology, stepwise refinement, distributed computing, message oriented programming, abstract data types, synchronization, resource, verification.

1. Introduction

In the last few years the various approaches to parallel programming, from Dijkstra's semaphores [10] to Hoare's monitors [17], responded to the need for 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 [22, 23], Jammel [19], Hoare [18] and in the description of multiprocessing systems such as Demos [1], Mininet [21] and Thoth [4].

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 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 this paper we attempt the development of a methodology for message oriented programming (i.e. programming with messages and processes) based

on the concept of resource. Resources may be viewed as generalizations to distributed environments of such concepts as monitors [3 , 17] and classes [9]. They may also be viewed as formalizations of concepts such as managers [19] and proprietors [4]. We believe that the concept of abstract data type is central in programming methodology and base our proposed methodology on them. However, just as for monitors and as opposed to the situation for sequential programs, we must also specify in some manner two important aspects of the use of an abstract data type. Firstly, we must specify how the operations of the data type may be invoked by users of the type. Secondly, we must specify the degree of parallelism which we wish to allow in the use of the operations of the type. Note, however, that the parallelism specified must not violate the properties of the type. We group these two aspects together under the name "synchronization". A resource is then an abstract data type together with these synchronization mechanisms.

We present in the next section an outline of and justification for our resource based methodology. The methodology is based on the concept of abstract data types plus synchronization together with the method of stepwise refinement. We may divide the methodology into two parts: the development of an abstract program and the refinement of this abstract program (by the use of more primitive constructs to implement the abstract constructs of the abstract program). The first part of the methodology has been more fully outlined and justified in [5 , 6 , 8] and we concentrate here on the second part. Section 3 presents the development of an example using the methodology. Section 4 discusses the principles of analysis which may be associated with the methodology to prove properties of the

programs. Proofs are also outlined for the example of section 3. Section 5 presents some concluding remarks and outlines possible directions for research in this area.

2. A Resource Based Methodology

We intend in this section to outline and motivate a methodology for the development of programs for distributed environments where the synchronization mechanism is based on the use of message passing primitives. The starting point for the development of our methodology can be encapsulated by the now famous aphorism which is the title of Wirth's book: "Programs = Algorithms + Data Structures". We modify this slightly (but significantly) by substituting "Abstract Data Types" for "Data Structures". The reasons for this have been stated and elaborated elsewhere [12, 13, 15, 20] and we will not repeat them here.

We note firstly, that the "Programs" of the aphorism were intended to identify sequential programs - i.e. programs which were run in a single processor environment. Similarly, the "Algorithms" were intended to be sequential algorithms.

By using the principle espoused, programs could be developed to solve particular problems by using a method called stepwise refinement. The first step in this method is to identify the data types which are appropriate for the solution of the problem. The appropriate control structures are then identified and an algorithm is developed which makes use of these abstract control structures and data types assuming that the control structures and data types are correctly implemented. The stepwise refinement principle now comes into effect and these control structures and data types are implemented (hopefully correctly) in terms of more primitive control structures and data types, respectively.

We would like to apply similar principles in a distributed environment. Thus a (non-sequential) program is defined in terms of a (non-sequential or distributed) algorithm and some distributed version of the concept of abstract data type. We want to retain the concept of abstract data type because of its demonstrated power in the development of programs. However, the concept was developed for sequential environments in which the operations of the type could be invoked by a sequential program - i.e. the operations could be invoked only one after the other. Thus the synchronization mechanism for the use of the operations of the data type is not explicitly specified as it is assumed to be a simple sequential one.

The methodology we develop will be based on the concept of resource. This concept has its roots in such notions as monitors [3 , 17] (essentially the development of a synchronization mechanism for the operations of a data type in a situation where a shared address space is assumed), managers [19], proprietors [4] and secretaries [11]. The underlying idea in all of these is the explicit expression of synchronization mechanisms. Resources are essentially an abstract data type together with a synchronization mechanism expressed in terms of message passing primitives. These mechanisms are used to allow as much parallelism as possible in the use of the operations of the data type. This is, of course, at the heart of the development of distributed computing - to gain efficiency by the use of parallelism. Thus for us Wirth's aphorism becomes:

(Distributed) Program =

(Distributed) Algorithms + Resources

where

Resource =

Abstract Data Type + Synchronization.

However, an aphorism is not worth anything unless its aptness can be demonstrated in a simple and convincing manner. We now outline our methodology and, hopefully, demonstrate the aptness of the above aphorisms with the example in the next section. For a fuller treatment of the first part of the methodology outlined below, see [5, 6, 8]. As for sequential programs, the first step in the development of a program is the identification of the abstract data types (i.e. the bases of the resources) to be used by the program. (As for sequential programs the choice will have a great influence on the final solution which is obtained.) Once an abstract data type has been identified (and presumably specified but not yet implemented) we are left with the task of adding the synchronization mechanism. This can be done as follows: Assume for the moment that the operations of the type can be directly invoked as primitive operations by the programs performing synchronization for the resource. Thus the resource is assumed to be implemented on a single address space. (This assumption will later be removed in the stepwise refinement process.) A formula, called the asynchronous condition formula (acf), is now developed which expresses the conditions under which operations of the data type may be invoked. Thus, it is at this point that one determines the degree of parallelism which is desired for the resource (within the limits set by the nature of the underlying data type). Using the acf, one can now define the process structure (i.e. the set of programs used for synchronization) associated with the resource based on one of two possible criteria. Firstly, the functional strength solution uses one process to control the invocation of the operations and thus results in a centralized and sequentialized form of synchronization. Secondly, the informational strength solution uses one synchronization process for each operation of the underlying type. Thus, this corresponds

to a distributed and (possibly) highly parallel solution. See also [5, 6 , 8].

Once the process structure for a resource has been defined, for each process in turn we establish a synchronization formula (sf) (in the form of a regular expression) which establishes the sequence of message passing primitives which the process will use to synchronize its activities with other processes in the resource (if there be such) and the environment of the resource. Then the process is defined by "filling in" the sequential parts of the process which "fit" between the synchronization primitives of the sf.

By following the above methodology, one obtains an abstract implementation of the resource and, based on the properties of the data type and the synchronization primitives, one can prove properties of the abstract program. At this point we use the idea of stepwise refinement and remove the assumption that the synchronization processes of a resource can invoke the operations of the underlying data type as primitive operations on a shared address space. Two situations are now possible. Firstly, we may keep the assumption of a shared address space for the processes and implement the resource's underlying data type in the usual way. This corresponds to the conventional situation and techniques for such implementations have been well developed and studied [12, 14, 15, 16]. Secondly, we may develop a distributed implementation for the data type. This involves the use of more primitive resources (as opposed to the data types of the conventional implementations) for the implementation of the resource in question. Thus, the direct invocation of a data type operation is replaced by a procedure call which implements the operation via a sequence of communications with

the resources being used for the implementation. Since all the synchronization of the operations of the data type is done now by message passing the common address space is no longer required and we may implement our distributed data type with a pool of processors with disjoint address spaces.

Thus in the next section we will present a solution to the well known bounded buffer problem. Our methodology will develop the resource called "n-bounded buffer" and then implement this resource in terms of the more primitive resources "Cell[i]" (a simple one place buffer) and "Pointer". Other solutions (both distributed and non-distributed) are also discussed.

The advantages to be gained from such an approach are similar to the benefits espoused for the analogous technique applied to sequential programs. That is, we gain modularity in the solution and we also modularise the proofs of properties of the system. The advantages of program modularity based on the use of abstract data types has been detailed elsewhere and we will not repeat these here. The modularity to be gained in proofs of properties will be more fully outlined in section 4, but we may point out here that since these programs use a control structure quite different from sequential programs, new techniques are needed for modularizing the traditional aspects of proofs (such as the correctness of implementation of a data type) as well as the new or novel aspects of proofs (such as deadlock freeness, lack of starvation, etc.). We hope our methodology lends itself to both these types of modularization.

Our examples will be written in an Algol or Pascal like language augmented by the following communications primitives:

$send_{\ell}(m,msg)$: 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 message m;

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

These primitives are part of a data type defined elsewhere [7]. We do not wish to say that the language illustrated is a "good" language, but we do not want to cloud the presentation of the methodology with a discussion of language issues.

3. An Example

To illustrate the resource based methodology described in the last section we will develop a distributed message-oriented programming solution to the bounded buffer problem. This problem can be informally 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 [18]. 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.

In this example, 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". We consider the producer and consumer processes as external resources which interact with the buffer-area resource. As mentioned briefly in the last section, the first part of the methodology (fully described in [6 , 8]) consists of (i) identification of the resources needed to solve the problem, (ii) establishment for each resource of the asynchronous condition formula (acf) to specify the conditions under which the operations may be invoked, (iii) establishment for each resource of the process structure by using the criteria of functional strength or informational strength together with synchronization formulas (sf's) which give the sequence of message passing primitives for each process that forms the resource, and (iv) "filling in" of the sequential part. The second part of the methodology, which is presented in this report, consists of the introduction of the stepwise refinement concept as explained above.

The resource identified for this problem is the buffer area. We assume that the data type (and consequently the operations "place an item" and "get an item") associated with the resource has been implemented. The conditions under which the operations "place an item" and "get an item" may be invoked are NF (buffer area not full) and NE (buffer area not empty), respectively. The corresponding asynchronous condition formula (acf) for the bounded buffer problem is expressed as follows:

$$NF \ \underline{or} \ NE \equiv \text{acf-producer} \ \underline{or} \ \text{acf-consumer}$$

(Note that if we had more than one producer or consumer process in the bounded buffer problem then the acf would be NF$\wedge$NW (no other writers) or NE$\wedge$NR (no other readers).)

We are going to use the informational strength solution of the bounded buffer problem as the starting point in the development of the ideas of our resource based methodology. In this solution, we associate with the resource one process for each of the preconditions "acf-producer" and "acf-consumer". (Each process will manage a particular aspect of the resource.) It is possible to base the solution of the problem on the process p-avpl(that manages the number of available places (avpl)) and the process p-avit (that manages the number of available items (avit)). The former process will treat the operation "place an item" from the producer if "avpl ≠ 0" (assuring NF) and the latter one will treat the operation "get an item" to the consumer if "avit ≠ 0" (assuring NE). The solution for the bounded buffer problem using the informational strength approach is given below. (Note that we omit a discussion of the sf's for these processes and the process of "filling in" as these are fully discussed elsewhere [6, 8].)

```
process p-avpl ;
   { t:  pair of strings ;
     c-avpl:  counter n ;
     area:  buffer ;
     while true do
          if cval(c-avpl) ≠ 0
          then { t := rec-any ;
                 case t.name of
                 { 'prod': {decr(c-avpl) ;
                            place(area,t.msg) ;
                            send(p-avit)}
                   'p-avit':  incr(c-avpl) }
          else { receive(p-avit) ;
                 incr(c-avpl) }
   }
```

```
process p-avit ;
   { v:  pair of strings ;
     c-avit:  counter 0 ;
     area:  buffer ;
     while true do
          if cval(c-avit) ≠ 0
          then { v:= rec-any ;
                 case v.name of
                 { 'cons': { decr(c-avit) ;
                             v.msg := get(area).value ;
                             send(cons, v.msg) ;
                             send(p-avpl) }
                   'p-avpl': incr(c-avit) }
          else { receive(p-avpl) ;
                 incr(c-avit) }
   }
```

(Note that the operations "place an item" and "get an item" can be executed concurrently in this informational strength solution.)

As mentioned in the last section, we assume that the process p-avpl and p-avit use a shared address space and thus are able to invoke the operations "place" and "get" of the data type called buffer directly. We have also used in the solution the data type called counter with the operations "cval" (gives the value of the counter), "incr" (increments the counter by 1) and "decr" (decrements the counter by 1). The definition of these data types is given as follows:

a) Type Counter-0

Operations:

init: → Counter

cval: Counter → Natmod n

incr: Counter → Counter

decr: Counter → Counter

Axioms:

cval(init) = 0

cval(incr(c)) = if cval(c) = n then error
else cval(c) + 1

cval(decr(c)) = if cval(c) = 0 then error
else cval(c) - 1

(Counter-n is essentially the same data type as above, but with the axiom cval(init) = n replacing the first axiom.)

b) Type Buffer

Operations:

new: → Buffer

place: Buffer x Value → Buffer

get: Buffer → Value x Buffer

x: → Value

Axioms:

We are assuming the following canonical form for a value of the sort Buffer:

place(place(...(place(new, x_1), x_2), ...), x_k).

get(new) = error

place(place(...(place(new, x_1), ..., x_n), x_{n+1}) = error

where n denotes the size of the buffer area

get(place(b, x)) = if b = new then <x, new>

else <first(get(b)), place(second(get(b)), x)>

where the operations "first" and "second" give back the first or the second element of the tuple respectively. Tuples are constructed by the operation <..., ...>.

(Example of using the last axiom:

get(place(place(new, a), b)) =

= <first(get(place(new, a))),

place(second(get(place(new, a))), b)>

= <first(<a, new>), place(second(<a, new>), b)>

= <a, place(new,b)>)

In the above solution, we assumed a centralized representation of the data (buffer area). At the next stage in the stepwise refinement process, this assumption can be changed to reflect a distributed representation of the data type. This is done by replacing operation invocations by appropriate message passing activities and the centralized version of the buffer by a distributed version. The resource buffer considered previously can be

replaced by (or defined in terms of) two resources: a resource called Cell [i] (belonging to a set of n identical cells) and a resource called Pointer (two versions of which are used as pointers to get items from the cells and to place items into the cells). The definition of the data types Cell [i] and Pointer is given as follows:

a) Type Cell[i]

Operations:

Λ: → Cell

read: Cell → Value

write: Value → Cell

x: → Value

Axioms:

read(Λ) = error

read(write(x)) = x

b) Type Pointer

Operations:

init: → Pointer

val: Pointer → natmodn

imodn: Pointer → Pointer

Axioms:

val(init) = 0

val(imodn(c)) = val(c) $\oplus$ 1

where $\oplus$ denotes addition modulo n.

We are going to use the first part of our methodology as described previously (and further detailed in [6 , 8] to design the programs that manage these two resources. We will also define the resource Counter assuming that we do not have direct access to it(that is, it belongs to another address space).

(i) Identification of the resources:

In this case, the resources are the Cell[i] and Pointer which form the distributed version of the buffer-area resource. The corresponding operations are "read" and "write" for the resource Cell[i], and "val" (value of the pointer) and "imodn" (increments the pointer by 1 modulo n). The resource Counter was defined above and, as we know, it includes the operations "val", "incr" and "decr".

(ii) Establishment of the acf:

The asynchronous condition formula (acf) for each resource, which expresses the conditions under which the operations of the data type may be invoked, is now developed. All the operations involved in the three resources Counter, Cell[i] and Pointer have similar characteristics, namely, the reading of the value of an object or the writing of a new value for the object. As we know, operations of reading and writing should be mutually exclusive. For example, where reading the value of a counter we should not increment or decrement its value and vice-versa. In view of this, the asynchronous condition formulas for the resources above are expressed as follows:

a) Counter : NI∧ND (condition for cval) or NR∧ND (condition for incr) or NR∧NI (condition for decr) where NR, NI, ND are the conditions "no reading", "no incrementing", and "no decrementing", respectively.

b) Cell[i]: NW (condition for read) or NR (condition for write) where NR, NW are the conditions "no reading", "no writing" respectively.

c) Pointer: NI (condition for val) or NR (condition for inmodn) where NI, NR are the obvious conditions.

(iii) Establishment of the process structure and sf:

In the functional strength approach, we associate with the resource one process which treats the conditions of the acf (separated by the delimiter or) by cases. In the informational strength approach, we associate with the resource one process for each of the conditions of the acf and therefore, each process will manage a particular aspect of the resource. When establishing the process structure of the resource, we can use the functional strength approach, the informational strength approach or a compromise between the two. A functional strength solution will be designed for the resources Counter and Cell[i] since it is not possible to achieve a higher parallelism by using more than one process to manage the resource. An informational strength solution will be chosen for the resource Pointer because the reading and writing operations are executed in different cells (slots of the buffer area).

Now for each process we write a synchronization formula (sf) which establishes the sequence of message passing primitives that the process will use to synchronize its activities with the other processes. A sf is a kind of regular expression with the usual delimiters ";", "or" and "*" (the symbols denote sequentiality, alternation and iteration). The synchronization formulas (which are trivial in this case) are defined below. Let us

denote receive by r and send by s.

a) Process c-avit: [r(p-avit); (s(p-avit, avit) or nil)]*

Process c-avpl: [r(p-avpl); (s(p-avpl, avpl) or nil)]*

b) Process cell-i: [(r(p-avpl) or (r(p-avit); s(p-avit, msg)]*

c) Process r-pointer: [r(p-avit); s(p-avit, first)]*

Process w-pointer: [r(p-avpl); s(p-avpl, last)]*

(When writing a sf, we use the other related and already designed expressions (sf's) in order to help the design of this sf or to validate the whole message passing activity (a complete match of the sending and receiving commands must exist). The expressions above will be used when designing the sequence of message passing primitives that will replace the direct invocation of operations in the centralized representation of the data type.)

(iv) Filling in of the sequential part:

This part is independent of the communication mechanism and it may depend on implementation details. We use the synchronization formulas as a skeleton to write the final form of the processes defined previously.

```
process c-avpl ;
   { avpl: integer;
     msg:  string ;
     avpl:= n ;
     while true do
        if (avpl < 0) v (avpl > n) then abort
            else { msg := receive(p-avpl) ;
                   case msg of
                   { 'r':  send(p-avpl, avpl) ;
                     'i':  avpl := avpl + 1 ;
                     'd':  avpl := avpl - 1 } }
   }
```

```
process c-avit ;
   { avit:  integer ;
     msg:   string ;
     avit := 0 ;
     while true do
        if (avit < 0) v (avit > n) then abort
            else { msg := receive(p-avit) ;
                   case msg of
                   { 'r':  send(p-avit, avit) ;
                     'i':  avit := avit + 1 ;
                     'd':  avit := avit - 1 } }
   }
```

```
process cell[i: 0..n-1] ;
   { x: pair of strings ;
     item:  cell-i ;
     while true do
        { x := rec-any ;
          if x.msg = nil
               then send (x.name, read(item))
               else write(x.msg) }
   }
```

```
process r-pointer ;
   { first:  pointer ;
     while true do
        { receive(p-avit) ;
          send(p-avit, val(first)) ;
          imodn(first) }
   }
```

```
process w-pointer ;
   { last:  pointer ;
     while true do
        { receive(p-avpl) ;
          send(p-avpl, val(last));
          imodn(last) }
   }
```

Now we are going to present the new code for the processes p-avpl and p-avit for the distributed solution. Here the direct invocation of a data type operation is replaced by a procedure call which implements the appropriate sequence of messages passing primitives for the new distributed implementation of the buffer-area resource.

```
process p-avpl ;
   { t:  pair of strings ;
     function  cval: integer ;
        { send(c-avpl, 'r') ;
          avpl := receive(c-avpl) }
     procedure incr ;
        { send(c-avpl, 'i') }
     procedure decr ;
        { send(c-avpl, 'd') }
     procedure place(item:  string) ;
        { send(w-pointer) ;
          send(cell[receive(w-pointer)],item) }
     while true do
         if cval ≠ 0
         then { t := rec-any ;
                case t.name of
                { 'prod': { decr ;
                            place(t.msg) ;
                            send(p-avit) }
                  'p-avit':  incr        }
         else { receive(p-avit) ;
                incr            }
   }
```

```
process p-avit ;
   { v:  pair of strings ;
     function cval:  integer ;
        { send(c-avit, 'r') ;
          avit := receive(c-avit) }
     procedure incr ;
        { send(c-avit, 'i') }
     procedure decr ;
        { send(c-avit, 'd') }
     procedure get(item:  string) ;
        { x:  string ;
          send(r-pointer) ;
          x:= receive(r-pointer) ;
          send(cell[x], nil) ;
          item:= receive(cell[x]) }
     while true do
          if cval ≠ 0
          then { v := rec-any ;
                 case v.name of
                 { 'cons':  { decr ;
                              v.msg := get ;
                              send(cons,v.msg) ;
                              send(p-avpl)     }
                   'p-avpl':  incr         }
          else { receive(p-avpl) ;
                 incr            }
   }
```

Other implementations of the buffer are of course possible. Firstly, assuming a non-distributed implementation, there are available the standard implementations of the resource's underlying data type. An obvious such implementation is in terms of an n-vector of objects and two integer pointers. Secondly, if we again assume a distributed implementation, a number of interesting solutions present themselves (including the one given above). For example, suppose we have available the resources cell[i] and counter, as above, and a new resource implementing the parameterized data type "queue of X". Then the buffer could be implemented by a queue of cells (of length less than or equal to n). The operation place could be implemented by checking to see if the buffer is full (using counter) and if it is not, then creating a new cell, placing the object in the cell and then inserting this cell in the queue (and of course, incrementing the counter). The operation get could be implemented by removing a cell from the front of the queue, if such exists, reading the object from the cell, destroying the cell and decrementing the counter. This implementation would be much more "dynamic" than the one presented above.

4. Verification of Resource Based Programs

It was indicated earlier that our methodology is such that it lends itself to the modularization of proofs of properties of the programs based on resources. That this is advantageous is clear from mathematics as well as experience with proving properties of sequential programs.

Before we can outline these proofs, however, we must define our correctness criteria. This is known to be a hard problem for distributed systems, but we hope that our methodology provides the framework for settling this issue for resource based programs. Proofs will be considered in two parts corresponding to the two parts of the methodology (abstract program and refinement).

Firstly, concerning the proofs for the abstract program, we have assumed in the first part of our methodology that the abstract data type on which the resource is based has been correctly implemented. Thus we may use the properties of the type in proofs concerning the "abstract" resource. We also have another criterion by which to judge the correctness of our implementation -- the asynchronous condition formula (acf). We thus propose to prove the correctness of our design by proving three properties:

(i) We must show that the acf guarantees the integrity of the underlying data type. That is, assuming operations are invoked only when the corresponding part of the acf is true, then the properties of the underlying data type are not "violated" by parallel invocations of operations.

(ii) We must show that the acf holds for the synchronization programs of the resource. More precisely, we must show that the part of the acf corresponding to a given operation is an invariant at the point in the synchronization program where the operation is invoked.

(iii) Finally, we must show some properties related to termination. For example, we must show that the resource is deadlock free with respect to its synchronization (communication) activities.

Secondly, we must show that the implementation of our abstract resources are correct. This amounts to showing that the resource's underlying data type has been correctly implemented in terms of other resources (which have already been proved correct). Techniques for this are fairly well developed as this amounts to showing that one data type is correctly implemented in terms of others.

The outline of the correctness proofs for the abstract programs are described as follows:

a) Resource Buffer:

Firstly, we are going to show that the asynchronous condition formula (acf) guarantees the integrity of the data type. As defined previously the acf for the resource Buffer is (NF or NE). This acf assures that we cannot get an item from an empty (new) buffer (first axiom of the data type) or put an item into a full buffer (second axiom of the data type). Therefore, it follows that the operations place and get can be invoked concurrently only if they refer to different slots of the buffer. (Otherwise, we have

an empty buffer or full buffer situation.) If the operations place and get work in different slots then the final configuration of the buffer will be equivalent to one of the sequences get(place(place(...(place(new, x_1),...), x_k), x_{k+1})) or place(second(get(place(...(place(new, x_1),...),x_k))),x_{k+1}) which correspond by the third axiom of the data type. Secondly, we have to show that the processes p-avpl and p-avit satisfy the conditions imposed by the acf. The process p-avpl assures that the operation place is only invoked when "avpl > 0" (NF). If "avpl = 0" then the process p-avpl blocks itself until the other process p-avit signals that a place is available. The same kind of reasoning can be used for the other case involving the operation get and the process p-avit which assures NE.

We have outlined proofs of the correctness of the abstract programs related to the conditions imposed by the acf. Now it is necessary to prove that the message passing activity does not lead to a deadlock situation. (Other properties such as starvation are not going to be treated here although our solution is starvation-free.) By analyzing the sequence of message passing primitives of the processes involved (producer, consumer, p-avpl and p-avit), we can see easily that there is a complete match of the sending and receiving operations in these processes. That is, every time a process x sends a message the corresponding process y is prepared to receive it. Therefore, the abstract program is deadlock-free. (For more details about the calculus for deadlock-freeness the reader should refer to [7].)

b) Resource Counter:

As we know, the acf for the resource Counter is ((NI∧NC) or (NR∧ND) or (NR∧NI)). This expression guarantees the integrity of the data type because the three conditions (separated by or) are handled by the same prccess and therefore the operations are executed sequentially (no possibility of reading and writing at the same time). For the same reason, it is easy to see that any of the processes p-avpl or p-avit satisfy the acf above. The simplicity of the communication mechanism (just one "send" and one "receive") makes the proof of deadlock-freeness trivial when relating this resource with the external environment.

c) Resource Cell[i]:

The acf for the resource Cell[i] is (NW or NR) as defined in the last section. The operations read and write of the data type Cell[i] are similar to the operations cval and incr (or decr) of the data type Counter. Here these two operations are also handled by the same process as in the previous case and thus, the operations read and write do not overlap. Deadlock-freeness also follows easily from the code of the program.

d) Resource Pointer:

In this case the acf for the resource Pointer is (NI or NR). The operations val and imodn of the data type Pointer have also a reading and writing behaviour, respectively. The whole reasoning about the correctness of the abstract program is similar to our last two cases. We are not going to repeat them here in order to save our readers from boredom.

The outline of the proof of correctness of the implementation of the buffer in terms of cells and pointers is as follows:

The initial buffer's properties are guaranteed by the initial values of Counter-0, w-pointer, and r-pointer (namely, 0) and that of counter-n (namely, n). For the axiom get(new)= error, we note that our synchronization program p-avit prevents the use of get on an empty buffer. Thus we have left out of the procedure get a check for this error. (To be strictly correct, we should rewrite this procedure as follows:

```
procedure get(item:  string) ;
   { x,y:  string ;
     send(r-pointer) ;
     x := receive (r-pointer) ;
     y := receive (w-pointer) ;
     if x = y then abort
     else { send(cell[x], nil) ;
            item := receive(cell[x]) }} ).
```

The correctness of place with respect to the second axiom re placing in a full buffer should be qualified by analogous comments to the above. That the last axiom can be verified is straightforward. The only other resource which has been (directly) implemented is counter and we leave it to the reader to verify its correctness.

5. Conclusions

This report presents a resource based methodology for the development of message oriented programming. As stated before, a resource is viewed as an abstract data type plus synchronization and may also be considered as generalizations to distributed environments of concepts such as monitors [17] and classes [9]. The principles of our resource based methodology are based on the concept of abstract data types. We may divide the methodology into two parts: the development of an abstract program and the corresponding refinement of this abstract program. The first part of the methodology involves the identification of the resources and the design of an abstract program assuming that the operations of the data types implemented by the resources can be invoked directly (fully justified in [6, 8]). The second part, which corresponds to the idea of stepwise refinement, was the main concern of this paper.

In section 2 our resource based methodology for distributed environments was motivated by referring to Wirth's famous aphorism "Programs = Algorithms + Data Structures". After that, it was outlined and justified by using the theory of abstract data types. In section 3 the methodology was illustrated by the design of several solutions (both distributed and non-distributed) for the bounded buffer problem. The solutions were fully described and exhaustively examined. In the last section we outlined a framework for proofs of the abstract programs where we made heavy use of the idea of modularization of proofs. We hope that the usefulness of the methodology has been demonstrated and we also think it may be used in other areas such as design of distributed data bases. Other directions for further research include: (i) a correction of the synchronization calculus ([7]) with some

appropriate calculus for the verification of program properties, (ii) use of the methodology in a dynamic environment with creation and destruction of resources, and (iii) specification of a language suitable for development of programs in our methodology. These questions will be the subject of future work.

6. 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 Nucleous of an Operating System; CACM, April 1970 (pp. 238-241, 250).

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

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

[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., Lucena, C.J., Maibaum, T.S.E.: A Methodology for Message Oriented Programming; Proceedings of the 6th GI Conference on Programming Languages and Program Development, Darmstadt, March 1980.

[7] 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.

[8] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: Message Oriented Programming - A Resource Based Methodology; Research Report, University of Waterloo, June 1980.

[9] Dahl, O.J., Myhrhang, B., Nygaard, K.: The SIMULA 67 Common Base Language; Pub. S-22, Norwegian Comptng. Ctr., Oslo, 1970.

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

[11] Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes; in Operating Systems Techniques, Academic Press, New York, 1972, (pp.72-93).

[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 RC6487, 1976.

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

[14] Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract Data Types and Software Validation; CACM, Vol. 21, No. 12, 1978, (pp. 1048-1064).

[15] Guttag, J.V., Horning, J.J.: Formal Specification as a Design Tool; Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL), 1980.

[16] Hoare, C.A.R.: Proof of Correctness of Data Representations; Acta Informatica, Vol. 1, No. 1, 1972, (pp. 271-281).

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

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

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

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

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

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

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