THE SPECIFICATION OF PROCESS SYNCHRONIZATION BY PATH EXPRESSIONS

R.H. Campbell
University of Newcastle Upon Tyne (G.B.)

A. N. Habermann
Carnegie-Mellon University, Pittsburg

Abstract

A new method of expressing synchronization is presented and the motivations and considerations which led to this method are explained. Synchronization rules, given by 'path expressions', are incorporated into the type definitions which are used to introduce data objects shared by several asynchronous processes. It is shown that the method's ability to express synchronization rules is equivalent to that of P and V operations, and a means of automatically translating path expressions to existing primitive synchronization operations is given.

12th December, 1973.

1. Introduction

The design and construction of the co-operation and co-ordination of concurrent processes is a difficult task, particularly in large operating systems. One major problem is the synchronization of actions belonging to separate processes. Some present methods, such as P and V [1] and Wait and Signal [2], are extremely primitive and subject to many hazards in programming. Other methods such as Monitors [3], Message Passing Systems [4] and Secretaries [5] are attempts to overcome some of these hazards.

We consider a process as operating, by a sequence of actions, on a known set of objects. Synchronization is required in order to maintain the integrity of objects which are shared between different processes. The explicit programming of critical sections and of communication of processes using P, V operations has the effect of spreading the implementation of synchronization of operations on shared data throughout the various programs of concurrent processes. Dijkstra's idea of the secretaries [5] controlling the operations on shared data can be viewed as a step in the direction of associating the specification of synchronization with the shared object. This paper proposes a general mechanism for representing synchronization rules in a consistent and coherent manner. It advocates the further step of combining these rules with "type definitions" that are used to introduce each class of object.

The new mechanism describes synchronization at the level of procedures. That is, if we want to synchronize two actions, each must be provided by a separate procedure invocation. The mechanism allows one to state what action sequencing is permissable, which is in direct contrast with synchronization schemes in which the main function is to prohibit or delay actions. Our proposal is that the synchronization be specified directly by describing how the body of one procedure, as a unit, is allowed to execute in relation to others, irrespective of when invoked by processes. The mechanism will specifically describe the synchronization permissible between executions of procedures and prohibit all others. That is, a process trying to execute one of the procedures must wait until the combination of circumstances specified in the synchronization has occurred.

The type definition we shall describe has two divisions; the first consists of the internal structure (for example data declarations and internal functions), the second, called its operations, consists of the procedures known to the outside program which are permitted to alter the internal structure. Together, the type definition and synchronization mechanism describe which procedures (operations) may be invoked by a program to access data objects and how these procedures are to be synchronized to allow the objects to be shared among separate processes.

First we shall describe our synchronization method and notation with several solutions of well known synchronization problems used as illustrations. Then we shall describe our notion of type and discuss, with the aid of examples, how it complements the synchronization method. Finally we shall show that our synchronization method is equivalent to P,V operations in terms of its ability to express synchronization, and describe an automatic method of translating our notation into existing primitive synchronization operations.

2. The Synchronization Mechanism

The proposed mechanism allows the synchronization between executions of procedures by separate processes to be specified by means of a path expression. Later we shall show how these path expressions may be used within a type declaration to describe the synchronization which will allow an object of that type to be shared by several processes. First, however, we shall identify what we think are fundamental synchronization schemes and describe how they may

be combined to describe complex synchronization.

The notation, although somewhat arbitrary, is based on regular expressions, which we feel provide a familiar framework and allows path expressions to be represented by finite state machines. We have also placed certain restrictions upon the notation which enable us later to describe an implementation of path expressions in terms of other synchronization primitives. We shall indicate the decisions we have made and attempt to justify them by example or by argument.

The idea underlying the mechanism can be envisaged as follows:- A path expression names the procedures whose execution by processes are to be synchronized. It includes a specification which describes exactly the way in which the synchronization is to be organized. Each path expression is implemented by a controller. Given that an individual synchronized procedure has been invoked by a process, the controller decides when the procedure execution should be allowed to commence, and therefore the process to continue.

The controller mechanism could operate as follows:- Each procedure commences with a prologue and finishes with an epilogue. A process executing the prologue of a synchronized procedure enquires of the ccntroller whether it may proceed. The controller, using the synchronization specification, may decide either to delay execution or to allow it to continue. Finally, when the process executes the epilogue of the procedure, it notifies the controller which may now be able to release other delayed processes.

The notation we have adopted is designed to simplify the construction of these controllers; we show that they might, for example, take the form of finite state machines constructed from P, V operations and semaphores [1].

The first two fundamental synchronization schemes we shall identify are the sequence of actions and the selection from a set of actions. (By action we mean the execution by a process of a procedure). A sequence of actions permits each one to occur in the order specified. Suppose the executions of three procedures p, q and r are to be sequentially synchronized. Then

p ; q ; r

is an example of a path expression which would, in our notation, express that procedures p, q and r are to be executed one after the other in the sequence given. The procedures may have been invoked by separate processes, in a different order and with possible intermediate delays. If an invocation of q occurs first, the invoking process will be delayed until procedure p has been executed. A selection from a set of actions permits only one to occur. Suppose the executions of the three procedures p, q and r are to be selectively synchronized.

Then

p, q, r

is an example of a path expression which would specify that a selection of one procedure is to be made from p, q and r. The process attempting to execute the procedure selected is allowed to continue, while processes attempting to execut those procedures not selected are delayed until a new selection is made from p, q and r. The selection of a procedure is made from amongst those procedures which have been invoked by processes. The selection is made using an unspecifie rule which ensures fair random order [1] (and which caters for any possible simultaneity).

These two basic schemes may be combined to form more complex path expressions.

Thus the path expression

p ; (q, r); s

synchronizes the executions of procedures, p, q, r and s. Executions of q and r are synchronized selectively. The executions of procedure p, the selected procedure from q or r, and the procedure s are synchronized sequentially, Thus the path expression permits two possible series of executions:- Either the execution of p precedes that of q which precedes that of s or the execution of p precedes that of r which precedes that of s.

For simplicity in the design of controllers for our notation, we allow a procedure name to occur once only in any path expression. This is not very restrictive because a procedure can always be renamed by embedding it inside another procedure.

There are two additional synchronization concepts which we find practical to represent. These are repetition and simultaneous execution.

Repetition permits a path expression once completed to be repeated. Many processes are cyclic in behaviour and this needs to be reflected in our synchronization notation. We represent repetition by enclosing a path expression between the key words path end. Again, for simplicity in design of controllers for our notation, we make the restriction that repeated path expressions may not be embedded within other path expressions. We have not found this restriction to be very important in the examples for which we have so far written synchronization rules and there are certainly ways in which this restriction may be relaxed. The path expression

path P1 end

synchronizes the procedure P1 so that it may be executed by processes repeatedly. If many processes invoke P1, one of them at a time will be allowed to execute P1 while the remainder are delayed until their turn comes. The path expression

path P1, (P2 ; (P3, P4)) end

is an example of a complex synchronization scheme involving the procedures P1, P2, P3 and P4. At first either procedure P1 or P2 may be selected to execute. If P1 is selected then, when execution by the process of P1 is complete, repetition will occur and a new selection made between procedures P1 and P2. If P2 is selected then, when execution by the process of P2 is complete, a selection will be made between procedures P3 and P4. In this case, when the procedure P3 or P4 which is selected has been executed by its invoking process, repetition will occur and a new selection made between procedures P1 and P2.

Simultaneous Execution permits several processes to execute given procedures concurrently. In many synchronization problems it is often desirable that a body of code can be simultaneously executed by several processes provided that by doing so the processes do not infringe other synchronization restrictions. The notation representing simultaneous execution is a bracket pair { } placed around a regular expression. These brackets may not be nested. One view of simultaneous execution is that it generates as many instances of the enclosed expression as there are requests for it until all instances have been completed. The path expression

{ P1 }

synchronizes the procedure P1 so that it may be executed by many processes simultaneously. Once one process begins to execute P1, other processes may do the same without delay, providing that there are outstanding (uncompleted) executions of P1. As soon as the last of these is finished, the path expression is considered to be complete and further processes invoking P1 will be delayed.

The path expression

path A ; { B ; C } end

synchronizes the procedures A, B and C using a combination of several basic synchronization schemes. Procedure A may be executed first by a process. On the completion of procedure A by a process, the sequence B ; C can be executed by many processes simultaneously. A process invoking procedure C will be delayed until the execution is completed by some other process of procedure B. Procedures B and C may be executed simultaneously by many processes, however the number of processes executing and which have executed procedure C can never exceed the number of processes which have executed B. If at some time all requests by processes for the sequence B ; C have been completed (the number of executed procedures B equals the number of executed procedures C and there are no more invocations of B) then repetition will enable a new invocation of A by a process to execute.

In our opinion, path expressions provide a clear and compact method for describing synchronization problems. For example, the path expression

path read, write end

specifies a series of executions by processes of the procedures read and write in unpredictable order, none of which overlap in time.

The path expression

path {read}, write end

specifies a series of executions by processes of the procedures read and write in unpredictable order. Read executions may overlap other read executions but write executions may not overlap other read or write executions. Reading, once started, will continue for as long as there are processes invoking read and at least one process executing read.

The above path specifications can be used, for instance, for programming file processing, and we shall now demonstrate how they may be adapted so that a particular access priority can be implemented and localized in one central place. In the last example, once reading commences, all processes requesting reading may proceed. It is therefore conceivable that one wants a policy in which, once writing commences, all processes requesting writing will proceed provided that they do so one at a time. This can be implemented by means of two path expressions.

path {read}, {WRITE} end

path write end

where WRITE is a procedure defined by

WRITE = begin write end

The internally defined procedure write actually performs the writing action. The first path ensures that if the read procedure begins to execute, all reading requests are accepted, and similarly with WRITE. The second path ensures that the actions of writing are mutually exclusive. Thus executions of WRITE are synchronized with respect to the first path and executions of its body (write) are synchronized with respect to the second path. The synchronization specification given by each path can be understood separately since, in the present proposal, a particular procedure name can appear in only a single path expression. (In other words there can be a separate controller with respect to each path).

Another strategy is to give individual read and write invocations by processes an equal chance of executing first. This can be achieved by having each process invoke a READ or a WRITE procedure which first obtains permission before performing the read or write. A construct with such properties is:

```
path  requestread, requestwrite end
path {openread; read}, write end
where requestread = begin openread end
      requestwrite = begin  write   end
      READ         = begin requestread; read end
      WRITE        = begin requestwrite end
```

The first path gives reading and writing an equal chance of starting. Once a read request has been granted, a read has been opened, thus the second path disables writing until the read has been executed. The braces in the second path make sure that reading can overlap if no writing is requested at all.

Suppose we require the strategy that writing should have priority over reading. This is the readers and writers problem solved by Courtois, Heymans and Parnas [9] and requires that when writing is requested, no further reading should be granted and writing should start as soon as the current reading is finished. The following construct has these properties:-

```
path  readattempt   end
path  requestread, {requestwrite} end
path {openread; read}, write end
where
      readattempt  = begin requestread end
      requestread  = begin openread end
      requestwrite = begin write end
      READ  = begin readattempt ; read end
      WRITE = begin requestwrite end
```

The purpose of the first path is to let only one read request occur at a time. While one process requests reading, all others have to wait until they can initiate a read attempt. This assures that a write request is granted in the second path at the earliest possible moment. The braces in the second path expression serve the purpose of immediately granting all write requests as long as writing is still going on. The braces in the third path allow, as before, read operations to overlap.

Finally, let reading have priority over writing, i.e., when reading is requested, from that moment no further write requests should be granted and reading should begin as soon as the current writing has been finished. A write request, on the other hand, should not stop the flow of reading. This can be written as:-

```
path  writeattempt  end
path {requestread}, requestwrite end
path {read} , (openwrite ; write) end
where  requestwrite = begin openwrite end
       writeattempt = begin requestwrite end
       requestread  = begin read end
       READ  = begin requestread end
       WRITE = begin writeattempt ; write end
```

The first path insures that only one prospective writer can request writing. The braces in the second path serve the purpose of letting all subsequent read requests through so that reading will not be interrupted by writing. The braces in the third path allow (as usual) an overlap in reading.

While not actually proving the correctness of our solutions above, we have been able to make assertions about their behaviour. The similarity between path expressions and finite state machines suggest that it might be possible to automatically prove or disprove assertions about solutions using them. These examples serve to demonstrate the synchronization facilities that path expressions provide and give an idea as to their power. We shall return to this point in the next section where it is shown that P & V operations can be simply programmed using path expressions.

3. Structuring Synchronization with Types

Types, under one name or another, have been used in programming languages for many purposes. Notable instances of their use are for checking as in Pascal [6], for describing building blocks as in extensible languages (for example ECL [11]) and for implementing new data objects and operations as in the classes of Simula 67 [8] or the modes of Algol 68 [7].

Programming languages have long provided an adequate tool for constructing complicated operations out of simpler ones by means of a procedure mechanism. An important aspect of such a tool from the designer's point of view is that it allows the programmer to separate the definition of an operation from its use. This has the advantage that, at the places where it is used, an operation can be treated as an object whose properties are known by its specification. Moreover, all implementation issues are now concentrated in one place and this facilitates validation and modifications of the implementation. We believe that type definitions should be used to accomplish on behalf of the construction of data objects the analogue of what procedure declarations do for operations. Thus, at the place where it is used, the details of the implementation of an object should be irrelevant and underlying structure should not be accessible at that moment. The reasons why this should be so are obviously the same as those that underly a procedure mechanism: separation of specification and implementation, and concentration of implementation issues so as to facilitate verification, debugging and modifications.

As a natural consequence of this point of view, a type definition is used to create new data objects which appear as atomic entities at the places where used. Drawing the parallel between type definitions and procedure declarations accessing a structural part of a typed object is similar to jumping into a procedure body.

The primary function of a type definition in a program is to describe the implementation of the operations on objects of this type in terms of earlier defined operations on the structural parts of such objects (though being the earliest of the three languages mentioned above, SIMULA 67 comes closest to this idea of type definition). Thus, another part of a type definition ought to be a description of the detailed structure that objects of this type will have.

These two functions of a type definition are reflected in a notation which we have devised to help us visualize our ideas. The syntax is purely arbitrary and is not intended as a proposal for a new programming language or any part of one.

For example we might have:-

```
type   buffer;
message   frame;
operations
   procedure  read  (returns message m) : m := frame;
   procedure  write (accepts message m) : frame : = m;
endtype
```

This example is a definition of a type of object called a buffer, whose structure consists of a variable frame of type message, and whose operations are the procedures read and write. (The type message is assumed to have been previously defined.) Instances of buffers can be declared or created in the scope of the type definition, and each one will contain its own instance of frame. The program using a buffer cannot, however, access frame directly but must use read and write. The procedures can be applied to them by means of the Simula 67 dot notation.

For example:

```
buffer A;
message T;
A. read (T);
A. write (T);
```

The type definition thus has two important properties:-

1. Protection of its structure by the scope rules.
2. Only a fixed, identifiable set of procedures is defined, giving carefully controlled access to the data of the objects of that type.

Objects created from type definitions can be common to the scope of two or more processes. The type buffer defined above is not satisfactory when various concurrent processes may simultaneously read and write a shared buffer and some form of synchronization is required.

In general, the sharing of objects of a type will be unsatisfactory if the data contained in the object can be corrupted by several processes executing procedures simultaneously or in invalid sequences. We will combine our path expressions with our notion of type to introduce some orderly structure in the sharing of objects.

The restrictions we must place on the operations read and write of our buffer example in order to preserve the integrity of the contained data are:-

1. Every read must be followed by a write.
2. Every write must be followed by a read.
3. A read and write must not execute simultaneously.

Provided the buffer obeys these rules we can assert that it will not lose or duplicate any information.

The following type definition ensures these three properties:-

```
type    buffer;
message frame;
path write; read end
operations
     procedure read (returns message m) : m : = frame;
     procedure write (accepts message m) : frame : = m;
endtype
```

The path is applied to the operations to produce the correct synchronization. A different instance of the synchronization path is associated with each instance of a buffer. Thus any declaration of a buffer will result in an atomic object which can be read or written alternately.

This type definition of a buffer may be used to build more complex data structures, for example objects of type "ring buffer". Suppose that a number of similar readers and writers wish to exchange information but are constrained by the amount of space available for buffers. One such device is demonstrated below and is designed to permit as much concurrency as possible. A ring of the above described buffers is declared. A send or receive request allocates a buffer to be read or written on a round-robin basis. The integrity of the buffers is assured by their type definition. Allocation is achieved by advancing pointers around the ring of buffers. Many requests to send or receive may occur simultaneously. However, each pointer may only be advanced by one process at a time if the integrity of the allocation mechanism is to be preserved. A type pointer is introduced which includes the necessary synchronization. Thus we have:-

```
type ring-buffer;
array 0 to N - 1 buffer R;
     type pointer;
     integer P = 0; path next end;
     operations
     procedure next (returns integer I):
          begin P: = (P + 1) mod N; I: = P; end;
     endtype;
pointer write-slot, read-slot;
operations
procedure send  (accepts message M):
     begin integer J; J: = write-slot.next; R[J].write (M); end;
procedure receive (returns message M):
     begin integer J; J: = read-slot.next; M: = R[J].read; end;
endtype;
```

The implementations of the buffers and the pointers are separated from the ring-buffer mechanism. Similarly the readers and writers can be programmed independently of the implementation of the buffering system.

The examples we have described above encourage us in our belief that the method is worth studying and is a potential contribution to better structured and safer synchronization methods. We shall now show that it is at least as powerful as the more primitive synchronization operations such as P and V [1]

or Signal and Wait [2]. For example the path path {V;P} end provides the synchronization necessary to implement P and V operations. The number of executed Ps can never be greater than the number of completely executed Vs. This path may be embedded within the type description for a semaphore. Thus:-

```
type semaphore;
path {V;P} end ;
operations
     procedure V : null;
     procedure P : null;
endtype;
```

Variables of type semaphore may be declared and each instance will have its own synchronizing path. The value of the semaphore can only be changed by executing either a P or a V. In the example above, semaphores are always initialized to zero. An extension to the above notation would be to include initialization in types and perhaps paths. Thus a program restricted to using our notation has lost none of the power of P, V operations but has gained the structuring facilities that the use of types and paths provide.

4. Implementation

One important aspect of our notation is that it has a practical implementation. Controllers for our notation can be implemented using existing synchronization methods and these may be generated automatically from the path expressions, for example by a compiler. We will show one particular implementation in which path expressions are transformed into appropriate P and V operations for use in the prologues and epilogues of the procedures the path expressions name. (Incidentally, this will complete the equivalence between the two synchronization methods.)

The following recursive algorithm will translate path expressions composed of the synchronization schemes of Section 2) above . Each path expression is subjected to repeated transformations, the final result providing the prologues and epilogues for each of the procedures named in the path expression. At each stage of the algorithm the path expression yet to be translated is labelled <pathexpression>. In general, the <pathexpression> will be surrounded by two generated synchronization operations O_L and O_R which are on its left and right respectively. The operation O_L may be either a P or a PP operation. (To simplify the algorithm two operations PP and VV are introduced which take three parameters, a counter and two semaphores. These operations will be explained later in terms of P and V.) The operation O_R may be either a V or a VV operation.

Stage 1) Select a unique semaphore S1, initialized to one, and

replace path <path expression> end by

P(S1) <path expression> V(S1)

Carry out stage 2) of the algorithm for <path expression>.

Finish.

Stage 2) Examine the <path expression> and depending upon the synchronization scheme of which it is composed do one of the following:-

a) A Sequence: The <path expression> is composed of:-

<path expression 1> ; <path expression 2>.

Select a unique semaphore S2, initialized to zero and replace the <path expression> by:-

<path expression 1> V(S2) P(S2) <path expression 2>

Carry out stage 2) for <path expression 1> and <path expression 2>

Finish of stage 2).

b) A selection: The <path expression> is composed of:-

<path expression 1>, <path expression 2>

Using the two synchronizing operations O_L (which may either be a P or a PP operation) and O_R (which may be either a V or a VV operation) enclosing the <path expression> replace the <path expression> using the replacement rule:-

O_L <path expression 1>, <path expression 2> O_R is replaced by

O_L <path expression 1> O_R O_L <path expression 2> O_R

Carry out stage 2) for <path expression 1> and <path expression 2>

Finish of stage 2).

c) Simultaneous Execution: The <path expression> is composed of:-

{ <path expression> }

Select a unique counter C1, initialized to zero, and semaphore S3, initialized to one. The operations O_L and O_R enclosing the <path expression> will be of the form

P(Si) { <path expression> } V(Sj)

Replace the operations and braces in the following way:-

PP(C1, S3, Si) <path expression> VV (C1, S3, Sj)

Carry out stage 2) for the remaining <path expression>.

Finish of stage 2).

d) Procedure name: The path expression remaining is just the name of one of the procedures to be synchronized. The synchronizing operation O_L (which may be a P or a PP operation) on the left of the procedure name is to be included in that procedure's prologue. The operation O_R (which may be a V or VV operation) is to be included in the epilogue of that procedure.

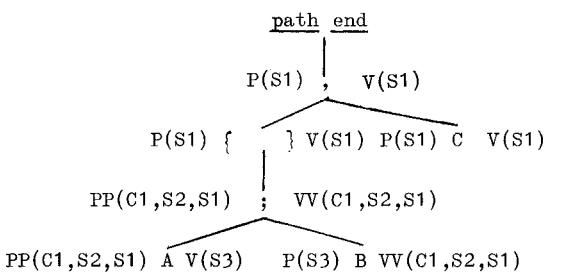
Finish of stage 2).

The operations PP and VV implement the simultaneous execution synchronization. Both operations share a counter C1 and a semaphore S3. The semaphore S3 is used to exclude more than one process from changing the counter at a time. The PP operation increments the counter, the VV operation decrements it. If the counter is increased from zero the operation P(Si) is invoked.(See below). If the counter is decreased to zero the operation V(Sj) is invoked.

```
procedure PP (counter C1; semaphore S3, Si); procedure VV(counter C1;semaphore
                                                                  S3,Sj);
  begin                                        begin
    P  (S3);                                     P (S3);
    C1 : = C1 + 1;                               C1 : = C1 - 1;
    if C1 = 1 then P(Si);                        if C1 = 0 then V(Sj);
    V (S3);                                      V (S3);
  end;                                         end;
```

The following example illustrates the translation of a path expression into P, V operations. The translation is represented by a tree. Each step of the algorithm corresponds to a node in that tree and the synchronization operations which have been generated up to a given step are written on either side of the corresponding node.

The path expression path ({A;B}), C end translates as:-



The resulting set of procedure prologues and epilogues which are created by part d) of the algorithm are written below in program form.

```
semaphore   S1 = 1, S2 = 1, S3 = 0; counter  C1=0;
procedures
      A :  begin  PP(C1, S2, S1) ; <body of A>; V(S3); end
      B :  begin  P(S3)          ; <body of B>; VV(C1,S2,S1) end
      C :  begin  P(S1)          ; <body of C>; V(S1) end
end
```

The procedures A, B and C implement the path precisely. Execution by a process of procedure A will set semaphore S1 to zero, (thus excluding the execution of procedure C by processes), semaphore S3 to one and the counter to one. Thus further processes may execute A and one process can execute B. When there are an equal number of executed A and B procedures and no further processes executing A the counter will have been reduced to zero and semaphore S1 set to one permitting the repetition of the path expression. If a process executes procedure C, it sets the semaphore S1 to zero to exclude processes from executing A. When a process finishes executing C it resets semaphore S1 to one allowing repetition of the path expression.

The example described above also serves to show that the path expression provides a structured synchronization technique which emphasizes what is needed,

not how it is to be achieved. The P, V implementation of the path expression does not directly express the synchronization it is used to create. (See also [10]).

Therefore, our mechanism can lead to automatically generated, well structured uses of synchronization primitives and the programmer is relieved of the problem of implementing his desired synchronization. Our choice of implementation is reflected in the constraints we have adopted in our notation. (See section 2). This aspect requires further investigation to ascertain what are the minimum set of constraints necessary to ensure unambiguous path expressions, and what are the minimum set of constraints for any given implementation.

Conclusion

We have introduced a new method of synchronization which provides a clear and structured approach to the description of shared data and the coordination and communication between concurrent processes. This method is equivalent to P and V operations with respect to its ability to express any given synchronization.

The path expression describes synchronization between executions of procedures by processes. It is a statement of all permissible synchronizations between the various procedures named within it. When combined with our type definition, it provides a powerful tool with which to design shared data objects. The type contributes the protection necessary to avoid carefully designed synchronization schemes from being upset by processes directly accessing the data and collects together in one place all the implementation details of a shared object. The path expression allows a specification of the synchronization needed to ensure the successful sharing of an object and does not require details of how that is to be done. Assertions can be made about the behaviour of path expressions. The resemblance of these expressions to finite state machines suggests that it may be possible to provide a means of automatically checking such assertions. Implementations of path expressions are possible using existing synchronization methods. In our notation we have restricted path expressions to allow for a simple implementation scheme, however this has seemed quite adequate for a variety of quite complicated synchronization problems. The algorithm which translates our notation into P and V operations has been used in a program to automatically generate code from path expressions. When the mechanism is used in this way the burden of implementing any given synchronization is removed from the programmer, eliminating the possibility of mistakes.

Acknowledgements

This work was carried out as part of R.H. Campbell's Ph.D. thesis at the Computing Laboratory of the University of Newcastle upon Tyne. We gratefully acknowledge the help of Dr. H.C. Lauer and Professor B. Randell for their help in preparing this paper. R.H. Campbell was financed by a grant from the Science Research Council.

References

[1] E.W. Dijkstra, Co-operating Sequential processes.
(in Programming Languages, F. Genuys, ed. Academic Press, New York, 1968).

[2] A.N. Habermann, Synchronization of Communicating Processes.
CACM 15, 3, (March 1972), pp. 171-176.

[3] C.A.R. Hoare, Monitors, An operating system structuring concept.
(To be published).

[4] P. Brinch Hanson, Nucleus of a Multiprogramming System. CACM 13, 4, (April 1970), pp. 238-241.

[5] E.W. Dijkstra, Hierarchical ordering of sequential processes. (in Operating Systems Techniques) ed. C.A.R. Hoare and R.H. Perrott, Academic Press. (1973).

[6] The programming language PASCAL , Acta Informatica, Vol. 1,1, (May 1971), pp. 35-63.

[7] Final Draft Report on the Algorithmic Language Algol 68.

[8] O-J. Dahl, B. Myrhhaug and K. Nygaard, the Simula 67 Common Base Language. Norwegian Computing Centre. (1970).

[9] P.J. Courtois, F. Heymans and D.L. Parnas, Concurrent Control with "Readers" and "Writers". CACM 14, 10 (October 1971). pp.667-668.

[10] P. Brinch Hansen, Structured Multiprogramming. CACM 15,7 (July 1972) pp.574.

[11] W. Wegbreit, B. Brosgol, G. Holloway, C. Prenner and J. Spitzen, E.C.L. Programmers Manual. Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts. (1972).