Conditions for the Equivalence of Synchronous and Asynchronous Systems

ERALP A. AKKOYUNLU, ARTHUR J. BERNSTEIN, SENIOR MEMBER, IEEE, FRED B. SCHNEIDER, MEMBER, IEEE AND ABRAHAM SILBERSCHATZ

Abstract—**Synchronous and asynchronous operation of software systems are defined. It is argued that certifying the correct operation of a system in the synchronous mode is significantly simpler than in the asynchronous mode. A series of compile-time and run-time restrictions for systems constructed in Concurrent Pascal are presented which assure equivalent operation in the synchronous and asynchronous modes.**

Index Terms—**Asynchronous processes, classes, Concurrent Pascal, concurrent processes, correctness, hierarchical operating systems, modularity, monitors, mutual exclusion, sequential operation, structured multiprogramming, synchronization.**

I. Introduction

As SOFTWARE SYSTEMS get increasingly complex, it has become more and more difficult to ensure their correctness. Attempts to deal with this problem range from the practical to the theoretical. On the practical side, techniques for improving the clarity of code (use of high-level languages, structured programming) and for modularizing programs have proved extremely valuable [6], [16], [8]. On the theoretical side, mechanisms for proving the correctness of programs are under development [10], [11].

Certification of systems that involve the concurrent activity of several processes is particularly difficult. This is because the behavior of the system in the concurrent mode of operation may be different than that obtained when each component process is run alone. In the concurrent mode of operation one asserts that each process will have a chance to execute (sometimes referred to as "finite progress"), but no specification is made as to the way concurrent process execution is interleaved or relative execution speeds. Even by employing the various synchronization mechanisms (e.g., Dijkstra's P and V [7], Brinch Hansen's Conditional Critical Regions [3], Monitors [3], [12]) it is possible for a process to see the system as a whole in a state that reflects partial execution of other processes. Such a state may not have been anticipated by the system designers, nor can it be easily reproduced.

The first proposals to control the complexity of a system were made by Dijkstra [6], [7]. He described a level structured approach to system construction in which a system can be implemented and debugged incrementally. This method produces a sequence of abstract machines gradually approaching the design target. The designer of the abstract machine implemented at some level need not be aware of the implementation details of the lower levels. Presumably, in such a system, each level will support one or more abstractions, which are then used by higher levels. The notion of "process" and its attendant synchronization primitives (P and V) is often defined at the lowest level of an operating system constructed in this manner. To guarantee that the operation of no level is dependent on the implementation details of a lower level, the information stored in a particular level is inaccessible to higher levels.

Parnas [16] stresses information hiding and implementation of abstractions by requiring that certain decisions made in the implementation of a module be hidden from other modules. Again, the system design task is simplified, because the designer need only be concerned with the properties of the abstraction, and not its implementation.

This notion of information hiding has been included in several high-level languages as abstract data types [5], [15] and has been adapted for use in asynchronous systems through the concept of a monitor [3], [12]. A monitor is a module that consists of a number of entry procedures, local procedures, and permanent variables. The latter are inaccessible to procedures external to the monitor. The only way to execute in the monitor is to call an entry procedure. Entry to a monitor is regulated by mutual exclusion so that at most one process is permitted to execute in the monitor at any time. This allows many processes to share the permanent variables in an orderly manner, and is an aid in guaranteeing their integrity. The **wait** statement is provided so that a process may suspend itself and relinquish control of a monitor in the event that the state of the monitor is not conducive to continued execution.

Hierarchical structuring of systems, abstract data types, and monitors are powerful tools for the design and construction of operating systems. They significantly reduce the complexity with which a designer must deal at any given time. Nevertheless, the designer must still understand the effects of parallel

Manuscript received June 19, 1978. This work was supported in part by the National Science Foundation under Grant MCS 76 04828.

E. A. Akkoyunlu is with the Department of Computer and Information Sciences, Brooklyn College, CUNY, Brooklyn, NY 11210.

A. J. Bernstein is with the Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794.

F. B. Schneider is with the Department of Computer Science, Cornell University, Ithaca, NY 14853.

A. Silberschatz is with Programs in Mathematical Sciences, University of Texas at Dallas, Richardson, TX 75080.

activity in the system in order to ensure that all interactions of concurrent processes are as intended. Such a task is considerably simplified if the sequence of computations within the system can be controlled so that specific orderings can be examined and reproduced. Unfortunately, this is frequently not possible in an operating system environment where sequencing depends upon factors (time quantum, device latency, cycle stealing) which are beyond the control of any user (debugger). Failure to anticipate interactions usually results in what are referred to as "time-dependent errors," caused by the sequence in which modules are executed by concurrent processes. The mutual exclusion and information-hiding properties of monitors are mechanisms for controlling access to the monitor's permanent variables. They do not, however, guarantee the consistency of the modules taken together. One would like to develop a programming methodology which ensures this consistency. Such a methodology was first described in a Ph.D. dissertation by Silberschatz [17], [18]. The systems discussed in that thesis were confined to those which did not contain **wait** or **signal** operations and several results proved here are analogous to results obtained there. This work extends those results to the case where synchronization primitives can be embedded in monitors.

An operating system implements a number of functions which can be called by the processes which it supports. The execution which results when a particular function is invoked by a particular process with specific parameters will be called a *request*. Thus two READ operations, even if they are invoked by the same process with identical parameters, will be considered two distinct requests. A request *terminates* if it completes execution and exits from the system. We define a *quiescent state* of the system to be one in which all requests which have been made by processes have terminated, and an *experiment* as the execution which occurs in the system in response to a set of system calls starting in a quiescent state and ending when all requests have either terminated or can progress no further (e.g., are waiting in monitors).

A large number of errors which may be present in an asynchronous system can be uncovered by testing the system in a purely sequential fashion. If we assume that a single request involves no asynchronous computation, then the system can be run sequentially if it is restricted to operate in such a way that it will not start processing a new request until the preceding one has either terminated or can progress no further. An experiment conducted in this way will be referred to as a *synchronous experiment.* Of course, if during the execution of a request it should awaken a previous request which had been suspended in the system, asynchronous activity will result and the experiment will not be synchronous. Note that since a synchronous experiment involves purely sequential operation, a synchronous experiment starting in a particular quiescent state always produces the same result. All other experiments involve the handling of two or more requests at the same time and will be called *asynchronous experiments.* If all synchronous experiments produce behavior acceptable to the designers of the system we will say that the system operates correctly in the synchronous mode.

Although asynchronism is essential in an operating system, arbitrary interactions between concurrently executing requests should be avoided. The results produced by a request should not depend on how its execution is interleaved with other requests or, more generally, on whether or not other requests are being serviced simultaneously. In this case, time-dependent behavior will have been eliminated and the outcome of an asynchronous experiment will be deterministic and reproducible. In this research we propose restrictions on the language used to implement a system which eliminates such time-dependent behavior. The restrictions guarantee that any result produced asynchronously can also be obtained by a synchronous experiment involving the same requests. Thus, if the system operates correctly in the synchronous mode, it will also function correctly in the asynchronous mode. This permits a designer to think of the processing associated with a request in isolation—as if no other activity were occurring in the system. Any interaction between concurrently executing requests is reduced to interaction between sequentially related requests. Debugging a system that exhibits this property is simplified since all timing-dependent errors, those errors that can occur only in the asynchronous mode of operation, are eliminated and the system need only be validated for synchronous operation to guarantee its complete correctness. Conversely, any error that occurs during asynchronous operation can always be reproduced while operating in the synchronous mode. It is our purpose in this paper to develop conditions for the equivalence of synchronous and asynchronous operation of a system, not to prove the correctness of the modules that comprise the system or the system as a whole.

The results presented here are described in terms of Concurrent Pascal [2]. This language provides for monitors. In addition its usefulness in the development of operating systems has been demonstrated in the implementation of several small systems [4].

II. An Example

We now illustrate the kind of problems that may appear in systems that do not exhibit equivalence between asynchronous and synchronous modes of operation by way of an example. A restructured version of the example, eliminating these errors, is given in Section IV.

Consider the following subsystem to be used for interprocess communication. In order to exchange "messages," a producer and consumer use a *mailbox*. The system supports NMAILBOX of these mailboxes in a monitor named MAIL_BOX_MON. Two operations are defined on a mailbox: READ_MSG, and WRITE_MSG, each performing the obvious operation. Mailboxes are designed for communication of single messages. A rendezvous facility is provided so that a producer may send a message to a consumer, though the system does not maintain a separate mailbox for each producer–consumer pair.

In order to acquire a mailbox, a producer calls the ALLOCATE function of the REN_MON monitor with a rendezvous number. The monitor reserves a mailbox for that producer and supplies the name of that mailbox as the value of the function. The producer may then call the WRITE_MSG

```
type ren_mon = monitor ;
    var rtbl : array [ 1 .. nmailbox ] of record
                                                inuse : boolean ;
                                                rid    : integer
                                            end ;
        no_in_use : 0 .. nmailbox ;
        notfull : condition { nmailbox > no_in_use } ;

function entry allocate ( rendezvous_id : integer ) : integer ;
    var entryloc : integer ;
    begin
        wait.notfull ;
        entryloc := 0 ;
        repeat
            entryloc := entryloc + 1
        until not rtbl [ entryloc ].inuse ;
        rtbl [ entryloc ].inuse := true ;
        rtbl [ entryloc ].rid     := rendezvous_id ;
        no_in_use := no_in_use + 1 ;
        allocate := entryloc
    end (* end of allocate *) ;

function entry reference ( rendezvous_id : integer ) ; integer ;
    var entryloc : integer ;
        foundit  : boolean ;
    begin
        foundit := false ;
        entryloc := 1 ;
        repeat
            if rtbl [ entryloc ].inuse and(rtbl [ entryloc ].rid = rendezvous_id)
                then
                    begin
                        reference := entryloc ;
                        foundit := true
                    end ;
            entryloc := entryloc + 1
        until foundit or ( entryloc > nmailbox ) ;
        if not foundit then reference := errorvalue
    end (* end of reference *) ;

procedure entry deallocate ( tblentry : integer ) ;
    begin
        rtbl [ tblentry ].inuse := false ;
        no_in_use := no_in_use - 1
    end (* end of deallocate *) ;

begin
            for no_in_use := 1 to nmailbox do
                begin
                    rtbl [ no_in_use ].inuse := false ;
                    rtbl [ no_in_use ].rid     := 0
                end ;
            no_in_use := 0
end (* end of ren_mon *) ;
```

Fig. 1. REN_MON.

procedure of MAIL_BOX_MON with this name and one message as parameters. The message is deposited in the named mailbox.

To retrieve a message from a mailbox, a consumer uses the REFERENCE function of REN_MON with a rendezvous number. Presumably, this rendezvous number is the same value that was used by the producer to obtain the mailbox in the first place. An error code ("errorvalue") is returned by this procedure if an attempt is made to reference a rendezvous number that is not currently allocated. The REFERENCE function returns the name of the mailbox that is associated with that rendezvous number, and the consumer uses this name in a call to the READ_MSG procedure of MAIL_BOX_MON to obtain the contents of the mailbox. After reading the message, the consumer calls the DEALLOCATE procedure of REN_MON with the mailbox name as a parameter to free the mailbox so that it can be used for other communications.

The protocol for message passing via mailboxes is then:

The producer:

1) Use REN_MON.ALLOCATE to obtain a mailbox.
2) Use MAIL_BOX_MON.WRITE_MSG to write a message.

The consumer:

1) Use REN_MON.REFERENCE to find correct mailbox name.
2) Use MAIL_BOX_MON.READ_MSG to read the message.
3) Use REN_MON.DEALLOCATE to free the mailbox.

Fig. 1 is an implementation of the REN_MON monitor, and Fig. 2 is an implementation of MAIL_BOX_MON. The automatic signal facility of Kessels [14] is assumed. The syntax

```
type mail_box_mon = monitor ;
   var mailboxes : array [1 . . . nmailbox] of mailbox ;
      i : integer ;
   function entry read_msg(mboxnme : integer) : message ;
      begin
         read_msg := mailboxes(mboxnme)
      end ; (*read_msg*)
   procedure entry write_msg(mboxnme : integer ; msg : message) ;
      begin
         mailboxes(mboxnme) := msg
      end ; (*write_msg*)
   begin
      for i := 1 to nmailbox do
         mailboxes(i) := nullmsg
   end (*mail_box_mon*)
```

Fig. 2. MAIL_BOX_MON.

and semantics of this conditional wait facility is described in Section III of this paper. Fig. 3 is an access graph [2] for the mailbox subsystem. A directed arc from one module to another indicates that the first may call the second during the execution of a request.

The subsystem described above does not function correctly under certain circumstances. The error that exists is time-dependent and therefore not one that would be observed by testing in a purely sequential fashion (i.e., every synchronous experiment produces a correct result). It requires the interaction of two requests in order to manifest itself. If requests from a producer and consumer are interleaved during asynchronous operation in such a way that the consumer completes its calls to REFERENCE and READ_MSG after the producer has completed its call to ALLOCATE, but prior to executing the call to WRITE_MSG, the consumer will receive the wrong message.

III. Discussion of the Results

Two major factors introduce time dependency in asynchronous operation. The first is illustrated in Fig. 4. It depicts a situation in which modules A and B, in processing results r_A and r_B, can each make calls on monitors M1 and M2. The following sequence of events can occur:

r_A calls A, executes in A;
r_A calls M1, executes in M1 and then returns to A;
r_B calls B, executes in B;
r_B calls M1, executes in M1 and then returns to B;
r_B calls M2, executes in M2 and then returns to B;
r_B exits B;
r_A calls M2, executes in M2 and then returns to A;
r_A exits A.

In this case, r_A sees the permanent variables of M1 prior to the time r_B accesses them, but sees the permanent variables of M2 after the time r_B accesses them. This is exactly the situation which occurs in the example in the previous section and one that could not possibly be reproduced in a synchronous experiment. It occurs when the execution of concurrent requests is interleaved in a particular way and is therefore timing dependent. It is to be hoped that the system designer has anticipated this situation. (A similar, time-dependent situation arises if it is possible for a request to enter the same monitor twice.)

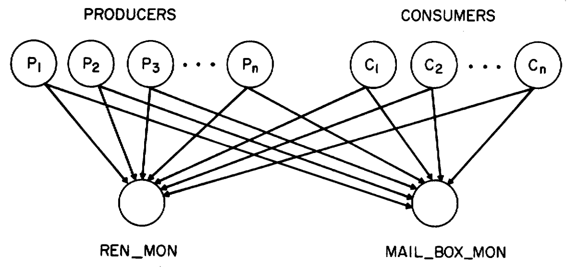


Fig. 3. Mailbox subsystem access graph.

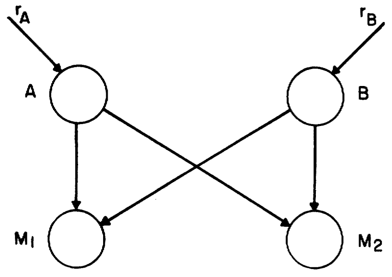


Fig. 4. Access graph which allows interleaving.

In order to avoid this type of interaction, we specify that a request may not enter a monitor after it has exited from a monitor (actually, a slightly different form of this restriction is used in the proofs given in the Appendix). This eliminates the interleaving described and it is shown in the Appendix that, as a result, all monitors entered by a request lie along a single path.

A second factor which introduces time dependencies during asynchronous operation involves the use of the **wait** statement. If a request can access the permanent variables of a monitor both before and after waiting in the monitor, then the request that waits and the request that causes the signal will be in a situation analogous to r_1 and r_2, as previously described. The signaler sees the permanent variables after partial execution of the waiter and the waiter sees them both before and after the signaler has executed. Although the monitor invariant is designed to guarantee that such a situation will always cause acceptable results, if it is not specified properly, errors will occur which cannot be reproduced in a synchronous experiment. In order to guarantee that each asynchronous experiment is equivalent to some synchronous experiment, a restriction on the use of the wait facility must be specified.

Various formulations of the wait facility have been described [13]. We specify the restriction in terms of the conditional wait facility as proposed by Kessels [14]. A programmer may declare one or more variables of type **condition** as part of the global declarations in a monitor. The syntax of such a declaration is

<condition name> : **condition** {<boolean expression>}.

Since the condition is declared global to the entire monitor (as opposed to being local to a particular monitor procedure), the boolean expression may only reference permanent monitor variables. Variables declared local to monitor procedures, and parameters to these procedures, may not appear in the boolean expression. A **wait** statement has the following syntax:

wait . <condition name>.

When executed, this statement causes the evaluation of the boolean expression referenced by <condition name>. If it is found true, the executing process continues; otherwise, the process is suspended on a queue associated with that condition.

A process is said to relinquish control of a monitor if it exits the monitor, or is suspended by a **wait** statement within the monitor. In the event that a process relinquishes control of a monitor, the conditions of any processes suspended in the monitor are evaluated, and, if one of these is true, that process is given control of the monitor. If none are true, then a process waiting outside the monitor may be given control. This scheme is essentially an automatic signal-on-return mechanism for synchronization. The order in which conditions are evaluated is left unspecified, provided no suspended process can wait on a true condition indefinitely. In addition, no primitives are provided to ascertain the number, or identities, of processes waiting on a **condition**.

Fig. 5 illustrates the use of this synchronization mechanism by implementing the single resource monitor found in [12].

The restriction we impose on the use of the wait statement is that if a wait statement appears in a monitor procedure it must be the first executable statement in that procedure. As a result a request cannot determine whether or not it actually suspended itself within a module and was awakened by another request. This is a limitation, and an attempt to weaken it is currently under study. This restriction can, however, be checked at compile time. The example in Fig. 5 complies with this restriction. Note also that due to the restriction the boolean expressions associated with conditions need only be tested when a process exits a monitor.

Using the two restrictions described in this section, it is possible to show (see the Appendix) that no request sees the permanent variables of any module in a state in which they have been partially updated by other requests. As a result, it is possible to show that there exists a partial order among the requests of an asynchronous experiment which is determined by the sequence with which they visit various modules, and it follows from this that every asynchronous experiment is equivalent to some synchronous experiment.

```
type single resource = monitor;
  var busy : boolean;
  inuse : condition {not busy};
  procedure entry acquire;
     begin
        wait.inuse;
        busy := true
     end;
  procedure entry release;
     begin
        busy := false
     end;
  begin
     busy false
  end;
```

Fig. 5. Single resource monitor.

IV. Example Revisited

It has been shown that certain asynchronous behavior (resulting in erroneous outputs) exhibited by the subsystem described in Section II is not reproducible in synchronous operation. Thus, although the system might function correctly when it is used in synchronous mode, this does not guarantee error-free behavior when it is run asynchronously. The source of the problem is that a request enters a monitor after it exits a monitor. As described in that section, REN_MON and MAIL_BOX_MON are monitors and both are called, during the execution of single requests, by the producer and consumer modules. In this section we present a mailbox subsystem which satisfies the restriction of previous sections. We are therefore guaranteed that, if the system functions correctly in synchronous operation, it will also function correctly asynchronously.

The access graph for the modified mailbox subsystem is shown in Fig. 6. Notice that producers (or consumers) no longer need to call two monitors, REN_MON and MAIL_BOX_MON, but rather call only one monitor, REN_MON, once. The communications protocol is simplified so that it is no longer necessary to explicitly allocate or deallocate a mailbox. A producer calls the RWRITE_MSG procedure of REN_MON with a rendezvous number and a message to be deposited. When space is available, a mailbox name is assigned and REN_MON calls MAIL_BOX_MON, using the assigned mailbox name. A consumer calls the RREAD_MSG procedure of REN_MON with a rendezvous number. If that rendezvous number is already in use (meaning that a message is already awaiting the consumer), then the correct mailbox name is determined by REN_MON, and is employed in a call to MAIL_BOX_MON; otherwise, an error value is returned. The modified implementation of REN_MON is shown in Fig. 7. The implementation of MAIL_BOX_MON remains as shown in Fig. 2, though it is now a **class** rather than a **monitor**.

Notice that the implementation is now in compliance with the restrictions, and thus equivalent operation in the synchronous and asynchronous modes will be observed. Since the error described in Section II was based on timing considerations during asynchronous operation, it can no longer occur in the modified subsystem.

V. Conclusion

In this paper a technique has been proposed for simplifying the design and validation of operating systems. By imposing certain restrictions on the language of implementation, it has been possible to eliminate time-dependent errors. Some of the restrictions can be checked at compile time; others by simple run time checks. The major limitation of this approach is the severity of the restrictions, and work is now in progress to relax them. Certain extensions are obvious. For example, since we have made no assumptions about the order in which processes suspended on a condition queue are awakened, it is possible to extend these results to include a priority wait mechanism [12]. Such a mechanism is discussed in [1]. Relaxation of the restriction on the way monitors may be called is being considered using managers [19]. The question as to whether actual operating systems can be constructed

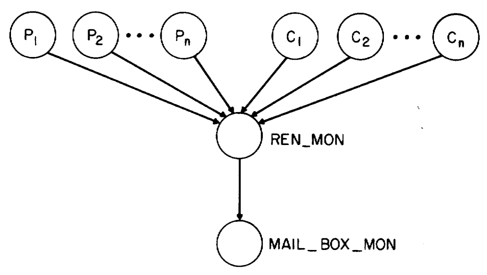


Fig. 6. Access graph of revised mailbox subsystem.

```
type ren_mon = monitor ;
   var rtbl : array [ 1 .. nmailbox ] of record
                                          inuse : boolean;
                                          rid   : integer
                                       end ;
      no_in_use : 0 .. nmailbox ;
      notfull : condition { nmailbox > no_in_use } ;

procedure entry rwrite_msg ( rendezvous_id : integer, msg : message ) ;
   var entryloc : integer ;
   begin
      wait.notfull ;
      entryloc := 0 ;
      repeat
         entryloc := entryloc + 1
      until not rtb1 [ entryloc ]. inuse ;
      rtbl [ entryloc ]. inuse := true ;
      rtbl [ entryloc ]. rid    := rendezvous_id ;
      no_in_use := no_in_use + 1 ;
      mail_box_mon.write_msg ( entryloc, msg )
   end (* end of rwrite msg *) ;

function entry rread_msg ( rendezvous_id : integer ) : message ;
   var entryloc : integer ;
      foundit  : boolean ;
   begin
      foundit := false ;
      entryloc := 1 ;
      repeat
         if rtbl [ entryloc ].inuse and(rtbl [ entryloc ].rid = rendezvous_id)
            then
               begin
                  rread_msg := mail_box_mon.read_msg ( entryloc ) ;
                  rtbl [ entryloc ].inuse := false ;
                  no_in_use := no_in_use - 1 ;
                  foundit := true
               end
         entryloc := entryloc + 1
      until foundit or (entryloc > nmailbox ) ;
      if not foundit then rread_msg := errormessage ;
   end (* end of rread msg *) ;

begin
   for no_in_use := 1 to nmailbox do
      begin
         rtbl [ no_in_use ].inuse := false ;
         rtbl [ no_in_use ].rid    := 0
      end ;
      no_in_use := 0
end (* end of ren_mon *) ;
```

Fig. 7. Revised implementation of REN_MON.

within these restrictions is currently being studied by the authors.

APPENDIX

A directed graph G is associated with each system. The nodes of the graph correspond to system modules, and an edge from module V_i to module V_j indicates that V_i can call V_j. Hierarchical systems are the only ones considered, and such systems are modeled by acyclic graphs. An example of a system graph is shown in Fig. 8. Note that the processes which initiate requests are not included. It is assumed that when a request is initiated, the process is blocked until the request terminates and control is returned. In what follows, the term path is used in the usual graph theoretic sense, and node-disjoint paths mean two paths having only their final node in common.

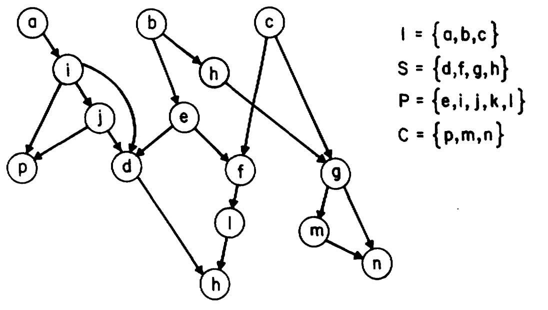


Fig. 8. A directed graph model of a system.

The nodes of G can be partitioned into four disjoint subsets:

1) I Nodes (interface): those nodes of in-degree 0.
2) S Nodes (shared): nodes reachable by at least two node-disjoint paths originating from two distinct I-nodes.
3) P Nodes (path): any node, except an S-node, which lies on a path from an I-node to an S-node.
4) C Nodes (common): the remaining nodes.

The various subsets for the system graph of Fig. 8 are also shown in the figure. I-nodes correspond to system modules which are the interface to process modules. Although it is possible that two process modules may call the same I-node, it is convenient to assume that every process is associated with a unique I-node (which may be functionally vacuous). S-nodes will be monitors and will therefore have a mutual exclusion mechanism which controls entry to them.

A request is initiated by a call from a process module to an I-node. Attention will be confined to requests obeying the following restrictions.

Restriction 1: A request involves no parallel processing (e.g., no **cobegin**, etc.).

Restriction 2: The system modules interact only through a call-return mechanism (i.e., no coroutine-type control is allowed).

Restriction 3: No module, on behalf of a request, may call more than one module of type S or P, and may only call such a module once.

The restriction "no request may call an S-node after it has returned from an S-node" is somewhat weaker than restriction 3 and can be used to derive the results presented here. Since the added flexibility is minimal and the proofs become more complicated, the stronger restriction is used. It should be noted that Eswaran *et al.* [9] derived a result for consistency in a multiaccess relational data base using this weaker form of the restriction. The result presented in this paper is similar to theirs in that it is concerned with maintaining the consistency of a data base—in the present case, the union of the data bases maintained by the monitors and classes in the system. However, language restrictions to guarantee this consistency are of concern here, and the operating system constructs that do not appear in a data base must be dealt with. The conditional **wait** statement is one such example.

The *subgraph induced by a request* is defined as that subgraph of G that contains all the nodes visited by the request and all the arcs traversed to visit these nodes.

In order to prove the equivalence of synchronous and asynchronous operation, one must proceed as follows. First, it is shown that under the restrictions all of the common modules visited by any two requests are all visited first by one request and then by the other (i.e., it is not the case that some common modules are visited first by one request while the remainder are visited first by the other request). Then it is shown that there exists a partial ordering on the requests so that there is at least one "last" request. This last request has the property that the execution of no request depends upon its outcome. The result then follows by induction on the number of requests.

To show that the common modules visited by two requests are visited in the same order by these requests one must take advantage of the mutual exclusion property of monitors. In the next lemma it is shown that the monitors visited by a request all lie on a single path. Then, in the bottleneck lemma, this result is used to establish that for every pair of requests that have nodes in common, there exists a unique common node, called the *bottleneck node*, through which both requests must pass prior to entering any other common module. Clearly, whichever request executes in the bottleneck node first will execute in all common modules first.

Lemma 1 (Unique-Path Lemma)

Let r_i be a request originating at interface node n and let G_i be the subgraph induced by r_i. Then G_i contains a path π which includes every S-node visited by r_i. Furthermore, if a and b are a pair of such S-nodes with a occurring before b on π, then every path from n to b in G_i contains a.

Proof: It is shown that if a and b are S-nodes visited by r_i, it is impossible to have two paths

$$\pi_a : n \cdots a, \quad b \notin \pi_a$$
$$\pi_b : n \cdots b, \quad a \notin \pi_b.$$

To show this, one must note that, since a and b are S-nodes, all predecessor nodes in π_a and π_b must be S- or P-nodes. But π_a and π_b are diverging paths which both originate at n. The node at which the two paths diverge would have to call two S- or P-nodes and this violates restriction 3. ■

Lemma 2 (Bottleneck Lemma)

Let G_i be the subgraph induced by a request r_i originating at the interface node n_i, and let G_j be the subgraph induced by request r_j originating at n_j, with $n_i \neq n_j$. Also, let C_{ij} denote the set of nodes common to r_i and r_j. If $C_{ij} \neq \phi$, then there exists a unique node $v_{ij} \in C_{ij}$ such that v_{ij} is an S-node, and for all $v \in C_{ij}$:

1) every path from n_i to v in G_i includes v_{ij};
2) every path from n_j to v in G_j includes v_{ij}.

Proof: Since $C_{ij} \neq \phi$, there is in G_i a path

$$\pi_i : n_i v_1 v_2 \cdots v_{k-1} v_k, \quad k \geqslant 1$$

such that $v_k \in C_{ij}$, and for all $h < k$, $v_h \notin C_{ij}$.

First it is noted that v_k is an S-node. This follows from the fact that $v_k \in C_{ij}$, there is in G_j a path π_j from n_j to v_k, and π_i and π_j are node-disjoint.

Next it is shown that v_k is the unique node which satisfies the requirements for v_{ij} stated in the lemma. Assume that for some node $v \in C_{ij}$ there is a path π_i from n_i to v in G_i which does not contain v_k. Let v' be the first node in π_i such that $v' \in C_{ij}$. By the same argument, v' must be an S-node. But this contradicts Lemma 1 since G_i has two S-nodes.

1) v' reachable from n_i through a path which does not contain v_k;
2) v_k reachable from n_i through a path which does not contain v'.

Thus, one can conclude that $v' = v_k$.

It has been established that in G_i all paths from n_i to any node $v \in C_{ij}$ contain the unique entry node v_k and that v_k is an S-node. Similarly, there is a unique S-node v_k' which is the entry node of all paths from n_j to $v \in C_{ij}$ in G_j. It can now be shown that v_k and v_k' must be the same node. Assume v_k and v_k' are distinct. There must be a path from v_k to v_k' in G_i since $v_k' \in C_{ij}$. Similarly, there must be a path from v_k' to v_k in G_j since $v_k \in C_{ij}$. This means that the overall graph G has a cycle. Thus

$$v_k = v_k'. \quad ■$$

Two additional restrictions required in order to prove the equivalence of synchronous and asynchronous experiments can now be stated.

Restriction 4: All **wait** statements are done via **conditions.**

Restriction 5: Any **wait** statement must be the first executable statement in a monitor procedure.

Restriction 5 partitions the execution of a request in a monitor procedure into two parts; the wait statement, and the state-changing part. A request will be said to have *entered* a module if it has executed statements other than a **wait** statement in the module. The actual execution of a particular request depends on the values of the permanent variables in each module entered. These values are affected by the previous requests, which have entered the module. For this reason the *precedes* relation, $<$, is defined. It will be said that $r_i < r_j$ (r_i precedes r_j) if r_i enters a module that r_j calls before r_j enters that module. (Note that, if r_j is forever blocked by the mutual exclusion code for that module, we shall still say $r_i < r_j$.) If $r_i < r_j$ or $r_j < r_i$, one can say that $r_i \rho r_j$, which means that r_i and r_j enter one or more common modules, or one enters a module and the other calls that module. The complement of this relation will be denoted by ρ'.

It will now be shown that the preceeds relation defines a partial ordering over the requests processed in a given experiment. This ordering is then employed to derive the synchronous experiment equivalent to the asynchronous experiment that induced the ordering.

Lemma 3 (Well-Ordering)

If $r_i \rho r_j$, then either $r_i < r_j$ in all common modules, or else $r_j < r_i$ in all common modules.

Proof: By Lemma 2, if $r_i \rho r_j$, then there exits a bottleneck node, v_{ij}, which is the first node that both r_i and r_j attempt to enter. Since v_{ij} is an S-node, it is a monitor. Let C_{ij} be the set of nodes entered by both r_i and r_j. It follows that if r_i enters v_{ij} before r_j, then r_j cannot enter any node in C_{ij} until after r_i returns (if it ever does) from v_{ij}. Note that after r_i returns from v_{ij}, it may not call any node in C_{ij} as a result of Restriction 3. Thus $r_i < r_j$ in all common modules. Similarly, if r_j enters v_{ij} before r_i, then $r_j < r_i$. ■

Lemma 4 (Sequencing Lemma)

Let r_x be an arbitrary request.

a) If r_x terminates, then for any pair, a, b of S-nodes visited by r_x, the event [r_x exits b] occurs after the event [r_x enters a].

b) If r_x does not terminate, then for any request r_y such that $r_x < r_y$, r_y does not terminate. Further, r_y is suspended at the bottleneck node of r_x and r_y.

Proof: Lemma 1 states that all S-nodes visited by a request are situated along a unique path. Let m_1, m_2 be two such nodes with m_1 occurring before m_2. It follows from Restriction 3 that there is no more than one call each to m_1 and m_2 in r_x. From the nested nature of calls, one has the sequence

$$[r_x \text{ enters } m_1] \cdots [r_x \text{ enters } m_2]$$
$$\cdots [r_x \text{ exits } m_2] \cdots [r_x \text{ exits } m_1].$$

Thus part a follows. To prove part b of the lemma, it is noted that in order for r_x not to terminate it must be blocked at some S-node, V_{blocked}. Also, $r_x < r_y$ implies that there exists a bottleneck node (which is an S-node), V_{xy}, which r_x has entered. This, in turn, implies that r_x is not blocked in V_{xy} and, therefore, $V_{xy} \neq V_{\text{blocked}}$.

It can be concluded that r_x entered V_{xy} before calling V_{blocked}. If V_{xy} is substituted for m_1 and V_{blocked} for m_2 in this sequence, then the events [r_x enters V_{blocked}], [r_x exits V_{blocked}], and [r_x exits V_{xy}] never occur. Since $r_x < r_y$, r_y called V_{xy}, but is blocked due to mutual exclusion. ■

Lemma 5 (Consistency Lemma)

For arbitrary requests $r_1, r_2, r_3, \cdots, r_n$, it is never the case that

$$r_1 < r_2 \wedge r_2 < r_3 \wedge \cdots r_{n-1} < r_n \wedge r_n < r_1.$$

Proof: Assume we have $r_1 < r_2, r_2 < r_3, \cdots, r_n < r_1$. Then there exist bottleneck nodes $v_{12}, v_{23}, \cdots, v_{n1}$. From the definition of $<$ it is known

r_1 enters v_{12}

r_2 enters v_{23}

$\vdots$

r_n enters v_{n1}.

By part b of the Sequencing Lemma, if any one of these requests is suspended they are all suspended. Thus there are two cases to consider.

Case 1: $r_1, r_2, \cdots, r_n$ are all suspended. Using Part b of the Sequencing Lemma, it can be stated that

r_1 enters v_{12} and r_2 calls but does not enter v_{12}

r_2 enters v_{23} and r_3 calls but does not enter v_{23}

$\vdots$

r_n enters v_{n1} and r_1 calls but does not enter v_{n1}.

Also, since r_1 enters v_{12} and is blocked at v_{n1}, one can conclude the following:

1) v_{12} and v_{n1} are distinct;
2) there is a path (v_{12} to v_{n1}).

Similarly, there exists paths (v_{n1} to $v_{n\,n-1}$), $\cdots$, (v_{23} to v_{12}). Since this implies that the graph contains a cycle, this case cannot occur.

Case 2: $r_1, r_2, r_3, \cdots, r_n$ all terminate. The following can be written:

assertion	*justification*
[r_1 enters v_{n1}] after [r_n exits v_{n1}]	since $r_n < r_1$
[r_n exits v_{n1}] after [r_n enters $v_{n-1\,n}$]	by Lemma 4
⋮	
[r_3 enters v_{23}] after [r_2 exits v_{23}]	since $r_2 < r_3$
[r_2 exits v_{23}] after [r_2 enters v_{12}]	by Lemma 4
[r_2 enters v_{12}] after [r_1 exits v_{12}]	since $r_1 < r_2$
[r_1 exits v_{12}] after [r_1 enters v_{n1}]	by Lemma 4.

The following contradiction has therefore been generated:

[r_1 enters v_{n1}] after [r_1 enters v_{n1}]

and the lemma is proved. ■

It is now shown that the execution of no request depends upon r_{max}, the last request in the partial ordering. Thus r_{max} can be run last in an equivalent synchronous experiment.

Lemma 6 (Deletion)

Let r_{max} be a request in an asynchronous experiment such that if r_i is any other request in that experiment, then ($r_i < r_{max} \vee r_i \rho' r_{max}$). Then the state of each module entered by r_{max}, at the time r_{max} enters that module, is the same as what it would be if r_{max} were not initiated until all other requests had either terminated or could progress no further.

Proof: Let r_{max} and r_i be as stated in the lemma and consider each r_i in the experiment. If $r_i \rho' r_{max}$, then r_i and r_{max} do not both enter any common module. Thus running r_i to completion before initiating r_{max} will not affect the state of any modules that r_{max} enters.

If $r_i < r_{max}$, then by Lemma 3, r_{max} can enter a module in $C_{i\,max}$ only after r_i has exited all modules in $C_{i\,max}$ for the last time. Furthermore, Restriction 5 ensures that prior to entering a module, a request can neither alter nor record in local variables the values of the permanent variables in that module. Thus, the state of all variables seen by r_{max} upon entering a module reflects the complete execution of r_i. ■

Requests, in addition to modifying variables in system modules, may make changes to the address space of the user module from which the request was initiated or cause some other action which is externally discernable (e.g., write a line to a terminal). Because such actions must be taken into account in deciding whether two experiments produce the same results, a *process pseudoprinter* to record them is defined. Associated with each process will be a pseudoprinter on which a line will be printed any time the system modifies the process address space or causes some other externally discernable action. The message written on the pseudoprinter will include all the particulars of the action (e.g., address changed and its new contents; line typed at terminal, etc.). The *system state* is defined as the values of all permanent variables in the system and the pseudoprinter output for all processes.

Two experiments are said to be *equivalent* if they involve the same requests, start from the same quiescent state, produce the same system state, and leave the same requests suspended. Notice that the present notion of equivalence concerns that which is visible to the program. The ordering, or contents of system queues, for example, is not included in system state, because this information is not available to a process. One can denote a particular synchronous experiment involving the successive initiation of requests $r_1, r_2, \cdots, r_n$ as $r_1; r_2; \cdots; r_n$.

Theorem

Any asynchronous experiment is equivalent to at least one synchronous experiment.

Proof: The proof is by induction on the number of requests in the experiment.

Base Case: Assume the experiment consists of two requests r_i and r_j (an experiment must contain two or more requests to be asynchronous). Without loss of generality, two situations must be considered, $r_i \rho' r_j$ and $r_i < r_j$. Using Lemma 6 one can conclude that in either case the state of each module entered by r_j, at the time r_j enters that module, is the same as what it would be after r_i had either terminated or could progress no further. Since the action taken by r_j is completely dependent on this state, it follows that the synchronous experiment r_i; r_j produces the same system state as the original asynchronous experiment.

Induction Case: Assume for each asynchronous experiment involving K or fewer requests there exists a synchronous experiment which produces the same system state.

Now consider an asynchronous experiment X which contains K + 1 requests. Let r_{max} be a request satisfying

$$(\forall r_i)[r_i < r_{max} \vee r_i \rho' r_{max}]. \tag{1}$$

Such an r_{max} exists due to Lemmas 3 and 5. Let X′ be the asynchronous experiment obtained from X by deleting r_{max}. By the induction hypothesis there exists $\hat{X}'$ which is a synchronous experiment equivalent to X′. Let $\hat{X}$ be the experiment obtained by running the synchronous experiment $\hat{X}'$, followed by r_{max}. (Thus $\hat{X} = \hat{X}'; r_{max}$.) Recall that a synchronous experiment is a sequence of requests where no request is submitted unless the preceding request terminated or could progress no further. Since $\hat{X}'$ is synchronous, $\hat{X}$ is synchronous because (1) guarantees that r_{max} cannot cause any previous request to awaken.

Let Y be the asynchronous experiment obtained by executing r_{max} after the completion of X′. Note that, due to Lemma 6 and the fact that the effect of r_{max} is determined solely by the system state it sees, Y must be equivalent to X.

Since $\hat{X}'$ and X′ are equivalent, r_{max} sees the same state if it runs following X′ or $\hat{X}'$. Thus it can be concluded that the experiment $\hat{X}'; r_{max}$ is equivalent to Y; $\hat{X}$ is therefore a synchronous experiment equivalent to X. ■

Acknowledgment

The authors wish to thank B. Ensor for his assistance in the development of the example in this paper.

References

[1] E. A. Akkoyunlu, A. J. Bernstein, and F. B. Schneider, "Medium term scheduling and equivalence of synchronous and asynchronous operation," Dep. Comput. Sci., SUNY at Stony Brook, Tech. Rep. 72, June 1977.

[2] P. Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199-206, June 1975.

[3] ——, *Operating System Principles.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[4] ——, "The solo operating system: A Concurrent Pascal program," *Software, Practice Experience*, vol. 6, pp. 141-149.

[5] O. J. Dahl, B. Myhrhaung, and K. Nygaard, *The Simula 67 Common Base Language.* Oslo, Norway: Norwegian Computing Center, 1968.

[6] E. W. Dijkstra, "The structure of THE multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 341-346, May 1968.

[7] ——, "Co-operating sequential processes" in *Programming Languages*, F. Genuys, Ed. New York: Academic, pp. 43-112.

[8] ——, "Notes structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Ed. New York: Academic, 1972.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 624-633, Nov. 1976.

[10] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Math* (vol. 19, American Mathematical Society, Providence, RI 1967), pp. 19-32.

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-687, Oct. 1969.

[12] ——, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 549-557, Oct. 1974.

[13] J. H. Howard, "Signaling in monitors," in *Proc. 2nd Annu. Conf. Software Engineering* (October 1976), pp. 47-52.

[14] J. L. W. Kessels, "An alternative to event queues for synchronization in monitors," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 500-503, July 1977.

[15] B. Liskov and S. Zilles, "An approach to abstraction," in *Proc. Symp. Very High Level Languages (SIGPLAN Notices*, vol. 9, Apr. 1974).

[16] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1053-1058, Dec. 1972.

[17] A. Silberschatz, "Correctness and modularity in asynchronous systems," Ph.D. dissertation, SUNY at Stony Brook, Aug. 1976.

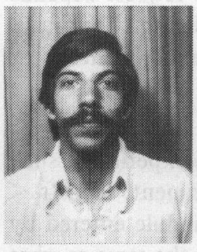
[18] A. Silberschatz and A. J. Bernstein, "Correctness in modular operating systems," Dep. Comput. Sci., SUNY at Stony Brook, Tech. Rep. 56, Nov. 1976.

[19] A. Silberschatz, R. B. Kieburtz, and A. J. Bernstein, "Extending Concurrent Pascal to allow dynamic resource management," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 210-217, May 1977.

Eralp A. Akkoyunlu, photograph and biography not available at the time of publication.

Arthur J. Bernstein (S'56-M'63-SM'78) received the Ph.D. degree from Columbia University, New York.

He has taught at Princeton University, and was a Research Scientist at the General Electric Research and Development Center in Schenectady, NY. He is now a Professor of Computer Science at the State University of New York at Stony Brook, specializing in the field of operating systems. His current interest is in concurrent programming and techniques for improving the reliability of asynchronous systems.



Fred B. Schneider (S'77-GM'78) received the B.S. degree from Cornell University, Ithaca, NY, and the M.S. and Ph.D. degrees from the State University of New York at Stony Brook.

He is currently an Assistant Professor of Computer Science at Cornell University, Ithaca, NY. His current research interests are in operating systems and programming languages, particularly concerning concurrency.

Abraham Silberschatz, for a photograph and biography, see this issue, p. 477.