CONDITIONS FOR THE EQUIVALENCE OF SYNCHRONOUS AND ASYNCHRONOUS SYSTEMS*

by

E.A. Akkoyunlu
A.J. Bernstein
F.B. Schneider
A. Silberschatz

State University of New York at Stony Brook
Department of Computer Science
Stony Brook, New York 11794

Technical Report #60

January 1977

1. Department of Computer Science, SUNY at Stony Brook, Stony Brook, N.Y. 11794
2. Department of Mathematics, University of Texas at Dallas, Richardson, Texas 75080

*This work was partially supported by NSF Grant GJ 28177.
I. INTRODUCTION

As software systems become 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 (e.g., the use of high level languages, structured programming) and for modularizing programs have proven to be extremely valuable [Dijk68a], [Parn74], [Dijk72]. On the theoretical side, mechanisms for proving the correctness of programs are under development [Floyd67], [Hoar69].

This problem is made an order of magnitude more difficult when a software system (e.g., an operating system) is designed for asynchronous operation. In this case, a given program can support several sites of activity or processes simultaneously and the results produced depend upon the relative timing of the execution of the various sites. The most productive approaches to date in dealing with the correctness problem as it applies to such systems have been in the areas of programming language design and system structure. Of course, all techniques which apply to synchronous systems apply here as well, but in this paper we will be concerned with techniques to control the added problems which arise from the asynchronous nature of the system.

The first proposals to go beyond the simple use of modularization to control complexity were made by Dijkstra [Dijk68a], [Dijk68b]. In order to limit the effects of timing on the results produced by asynchronous systems he described semaphores and their use by processes which must share data, and techniques for structuring the modules in a hierarchical fashion so that a system could be implemented and debugged incrementally. In the
hierarchical approach a data base implemented at a particular level is inaccessible to all modules at higher levels. Parnas [Parn74] stresses this point further by emphasizing that certain decisions taken in the implementation of a module be hidden from other modules. This notion of modularity has been utilized in high level languages in the development of the concept of abstract data types using classes [Dah168] and clusters [Lisk74]. Here a user-defined type is implemented as a module which contains both a description of an instance of the type and all the procedures which access it. An instance of a type is represented by a collection of permanent variables (i.e. Algol own) declared within the module whose values therefore persist between calls to procedures in the module. The procedures are callable from other modules and function as high level operations on the instance. This effectively divides the system into two regions, an outer region in which program components see an instance of the type as a primitive object which can be manipulated only by calls to the procedures provided for the type and an inner region within the type module in which design decisions on how the type is to be implemented are hidden.

This approach to modularity forms a useful framework with which to build the components of an asynchronous system. Here, in addition to creating abstract data types, modules can be used to implement algorithms which provide access to resources shared by several processes (e.g. disk, shared files). Since such modules may be simultaneously called on behalf of these processes they must contain a mechanism to provide mutual exclusion in addition to the features associated with classes, namely a private data base and procedures for accessing it. The monitor [Brin75], [Hoar74] has been proposed for this purpose. Mutual exclusion occurs automatically on monitor entrance and exit and queue variables are provided for use within
a monitor so that a process may put itself into a wait state pending the release of a resource by another process. Thus a monitor has the form shown in Figure 1.

type <monitorname> = monitor
  var (*declaration of permanent variables private to monitor*);
  procedure entry <abstractop1>(...);
    begin (*procedure body implementing an abstract operation*)
      ...
    end;
  procedure entry <abstractop2>(...);
    begin (*procedure body implementing an abstract operation*)
      ...
    end;
    ...
    begin (*initialization*)
      ...
    end;
end;

Figure 1. Structure of a monitor

Monitors have been implemented in the language Concurrent Pascal [Brin75] and a small operating system has been written [Brin76]. The operating system consists of a collection of classes and monitors which call each other in a hierarchical way. User code is executed in special modules called process modules. Each process module serves as the initial site of activity for what is commonly known as a process. The system is invoked when a process executing in a process module makes a call to some monitor or class of the system. The process will then execute for a period
of time within one or more system modules, perhaps waiting on queue
variables, and will ultimately return to the process modules when the
requested service has been performed. A nested call-return mechanism is
used for transfer of control between modules.

Although concepts such as hierarchical structuring of modules, abstract
data types and monitors are powerful tools which can be used in the con-
struction of an operating system, they do not guarantee correctness. The
purpose of the research described here is to develop a methodology for
building correctly functioning asynchronous systems. This methodology
was first described in a Ph.D. dissertation by Silberschatz [Silb76a],
[Silb76b]. The systems discussed in that thesis were confined to those
which did not contain \textit{wait} or \textit{signal} operations and several results proven
here are analogous to results obtained in that thesis. This paper extends
those results to the case where synchronization primitives can be embedded
in monitors.

In order to describe the approach taken it is necessary to define
several terms. A system implements a number of functions which can be
invoked by user processes (e.g., read a file, send a message). The invoca-
tion of such a function is referred to as a \textit{system call}. The execution which
results when a particular function is invoked by a particular process with
specific parameters will be called a \textit{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 \textit{quiescent state} of the
system to be one in which all system calls 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.

The results produced by a programming system supporting asynchronous computations depend upon the order in which these computations are performed. It is the job of the programmer to make sure that all results which can be produced are acceptable. 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).

Consider a system in which calls for service from several processes are being handled at the same time and involve the execution of common system modules. Even if these modules were monitors and the mutual exclusion associated with these monitors prevented more than one process from executing in the monitor at the same time, the processes might still interact in a destructive way. If for example request $r_1$ and $r_2$ both call upon modules $M_1$ and $M_2$ and if execution proceeds as shown in Figure 2 (where $t_{i+1} > t_i$), then neither $r_1$ nor $r_2$ views the union of the data bases contained in $M_1$ and $M_2$ at a time when the full effect of the other has been achieved. This may result in an error.
<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$r_1$ enters $M_1$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$r_2$ enters $M_1$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$r_2$ enters $M_2$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$r_1$ enters $M_2$</td>
</tr>
</tbody>
</table>

Figure 2. An example of the interaction of two requests

Such an error depends on the sequence of execution shown in Figure 2, and would not have occurred if $r_1$ had entered $M_2$ prior to the entry of $r_2$ in $M_1$.

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 in a sequential mode if it is restricted to operate in such a way that it will not start processing a system call 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 awake a previous request which had been blocked in the system, asynchronous activity will result and the experiment will not be synchronous. A particular synchronous experiment involving the successive initiation of requests $r_1; r_2; \ldots; r_n$ will be denoted $r_1; r_2; \ldots; r_n$. 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. Clearly it would be impossible to actually perform all synchronous experiments but since no time dependent errors can occur in this mode of operation it follows that locating an error is much simpler than when the system operates asynchronously.

Because of the simplicity of synchronous, as compared with asynchronous operation one would like to limit the debugging process to the former mode of operation. Using this reasoning we will assume that the system has been debugged for synchronous operation and therefore the outcome of any synchronous experiment will be assumed correct. In what follows we propose several restrictions on the language used to implement the system and prove that if these restrictions are adhered to, the results produced by any asynchronous experiment can also be obtained by a synchronous experiment involving the same system calls. Thus correctness in the synchronous case implies correctness in the asynchronous case. Note that a deadlock occurring in an asynchronous experiment must also occur in some synchronous experiments.
II. GRAPH MODEL OF A SYSTEM

We associate a directed graph $G$ 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 we consider, and such systems are modeled by acyclic graphs. An example of a system graph is shown in Figure 3.

![Graph of a system](image)

$I = \{a, b, c\}$

$S = \{d, f, g, h\}$

$P = \{e, i, j, k, l\}$

$C = \{p, m, n\}$

Figure 3. A directed graph model of a system

Note that process modules are not included. We will assume that when a process module initiates a request it is blocked until the request terminates and control is returned.

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

1. $I$ - nodes(interface): those nodes of indegree 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 Figure 3 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. We will confine our attention to requests obeying the following restrictions:

**Restriction 1.** A request involves no parallel processing.

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

**Restriction 3.** No S-node is called more than once in the same request.

**Restriction 4.** No module may call more than one S or P-node in the same request.

The restriction "no request may call an S-node after it has returned from an S-node" is somewhat weaker than the combined effect of restrictions 3 and 4 and can be used to derive the results presented here. Since the added flexibility is minimal and the proofs become more complicated we will use the stronger restrictions.

We define the *subgraph induced by a request* $r_i, G_i$, as that subgraph of $G$ that contains all the nodes visited by $r_i$ and all the arcs traversed to visit these nodes. The lemmas in this section develop some properties of subgraphs and requests which will be used later to relate synchronous and asynchronous experiments.
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 \( \pi \) 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 \( \pi \), then every path from \( n \) to \( b \) in \( G_i \) contains \( a \).

Proof: We show that if \( a \) and \( b \) are S-nodes visited by \( r_i \) it is impossible to have two paths:

\[ \pi_a : n \ldots a, \ b \not\in \pi_a \]
\[ \pi_b : n \ldots b, \ a \not\in \pi_b. \]

To show this we note that since \( a \) and \( b \) are S-nodes all predecessor nodes in \( \pi_a \) and \( \pi_b \) must be S- or P-nodes. But \( \pi_a \) and \( \pi_b \) 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 4.

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 \( C_{ij} \neq \emptyset \), then there exists a unique node \( v_{ij} \in C_{ij} \) such that \( v_{ij} \) is an S-node, and for all \( v \in C_{ij} \),

i. every path from \( n_i \) to \( v \) in \( G_i \) includes \( v_{ij} \), and

ii. every path from \( n_j \) to \( v \) in \( G_j \) includes \( v_{ij} \).

Proof: Since \( C_{ij} \neq \emptyset \), there is in \( G_i \) a path

\[ \pi_i : n_i v_1 v_2 \ldots v_{k-1} v_k, \ k \geq 1 \]

such that \( v_k \in C_{ij} \), and for all \( h < k \), \( v_h \notin C_{ij} \).
First we note 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 \( \pi_j \) from \( n_j \) to \( v_k \), and \( \pi_i \) and \( \pi_j \) are node-disjoint.

Next we show 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 \( \pi_i \) from \( n_i \) to \( v \) in \( G_i \) which does not contain \( v_k \). Let \( v' \) be the first node in \( \pi_i \) 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:

- \( v' \) reachable from \( n_i \) through a path which does not contain \( v_k \), and
- \( v_k \) reachable from \( n_i \) through a path which does not contain \( v' \).

Thus, we can conclude that \( v' = v_k \).

We have 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 \( G_j \). We now show that \( v_k \) and \( v'_k \) must be the same node. Assume \( v_k \) and \( v'_k \) are distinct. There must be a simple path from \( v_k \) to \( v'_k \) in \( G_i \) since \( v_k \in C_{ij} \). Similarly there must be a simple 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. The assumption is contradicted and \( v_k = v'_k \).
Lemma 2 establishes that for every pair of requests that have nodes in common, there exists a unique common node through which both of the requests must pass prior to entering any other common node. We will refer to such a node as a bottleneck node.

III. DEMONSTRATION OF EQUIVALENCE

In this section it will be shown that systems can be designed having the property that results produced by any asynchronous experiment can also be obtained with a synchronous experiment.

We assume the existence of the conditional wait facility as proposed by Kessels [Kess76]. 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>}

Due to the placement of the declaration of a condition, 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 of these outstanding conditions is true, then a process waiting outside the monitor is given control. This scheme is essentially an automatic signal-on-return mechanism for synchronization. The order that 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.

Figure 4 illustrates the use of this synchronization mechanism by implementing the single resource monitor found in [Hoare74].

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;

Figure 4. Single Resource Monitor
We now state two additional restrictions required in order to prove the equivalence of synchronous and asynchronous experiments.

**Restriction 5.** All \textit{wait} statements are done via \textit{conditions}.

**Restriction 6.** Any \textit{wait} statement must be the first executable statement in a monitor procedure.

As a result of Restriction 6, a request can not 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 Figure 4 complies with this restriction.

Restriction 6 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 \textit{entered} a module if it has executed statements other than a \textit{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 in the module. For this reason the \textit{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 $r_j < r_i$, we will 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 $\rho'$. 
The precedes relation has some properties which shall now be demonstrated and employed subsequently.

Lemma 3 (Well-Ordering)

If \( r_i \prec r_j \) 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 \( r_i \prec r_j \), 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 \( [r_x \text{ exits } b] \) occurs after the event \( [r_x \text{ enters } a] \).

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 \( m_2 \). Restriction 3 states that there is no more than one call each to \( m_1 \) and \( m_2 \) in \( r_x \). It follows from the nested nature of calls that we have the sequence

\[
[r_x \text{ enters } m_1] \ldots [r_x \text{ enters } m_2] \ldots [r_x \text{ exits } m_2] \ldots [r_x \text{ exits } m_1]
\]

Thus part a follows.

To prove part b of the lemma, we note that in order for \( r_x \) not to terminate it must be blocked at some S-node, \( V_{\text{blocked}} \). 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 \( V_{xy} \neq V_{\text{blocked}} \).

We can conclude that \( r_x \) entered \( V_{xy} \) before calling \( V_{\text{blocked}} \). If \( V_{xy} \) is substituted for \( m_1 \) and \( V_{\text{blocked}} \) for \( m_2 \) in the above sequence then the events \([r_x \text{ enters } V_{\text{blocked}}]\) and \([r_x \text{ exits } V_{xy}]\) 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 \( r_i, r_j, r_k \), it is never the case that

\[
(r_i < r_j) \land (r_j < r_k) \land (r_k < r_i).
\]
Proof: Assume we have \( r_i < r_j \), \( r_j < r_k \), \( r_k < r_i \). Then there exist bottleneck node \( V_{ij} \), \( V_{jk} \), and \( V_{ki} \). From the definition of \(<\) we know:

- \( r_i \) enters \( V_{ij} \)
- \( r_j \) enters \( V_{jk} \)
- \( r_k \) enters \( V_{ki} \).

By Part b of the Sequencing Lemma, if any one of the three requests is suspended, they are all suspended. There are thus 2 cases to consider.

Case 1: \( r_i \), \( r_j \), \( r_k \) are all suspended. Using Part b of the Sequencing Lemma, we state:

- \( r_i \) enters \( V_{ij} \) and \( r_j \) calls but does not enter \( V_{ij} \)
- \( r_j \) enters \( V_{jk} \) and \( r_k \) calls but does not enter \( V_{jk} \)
- \( r_k \) enters \( V_{ki} \) and \( r_i \) calls but does not enter \( V_{ki} \).

Also, since \( r_i \) enters \( V_{ij} \) and is blocked at \( V_{ki} \), we conclude

(i) \( V_{ij} \) and \( V_{ki} \) are distinct
(ii) there is a path \((V_{ij} \text{ to } V_{ki})\).

Similarly, there exists paths \((V_{ki} \text{ to } V_{jk})\) and \((V_{jk} \text{ to } V_{ij})\). Since this implies that the graph contains a cycle, this case cannot occur.

Case 2: \( r_i \), \( r_j \), \( r_k \) all terminate. We can write:
assertion  
[r_i enters V_{ki}] after [r_k exits V_{ki}] since r_k < r_i  
[r_k exits V_{ki}] after [r_k enters V_{jk}] by Lemma 4  
[r_k enters V_{jk}] after [r_j exits V_{jk}] since r_j < r_k  
[r_j exits V_{jk}] after [r_j enters V_{ij}] by Lemma 4  
[r_j enters V_{ij}] after [r_i exits V_{ij}] since r_i < r_j  
[r_i exits V_{ij}] after [r_i enters V_{ki}] by Lemma 4  

We therefore have generated the contradiction  
[r_i enters V_{ki}] after [r_i enters V_{ki}]  

and the lemma is proven.

Lemma 6. (Deletion)

Let \( r_{\text{max}} \) be a request in an asynchronous experiment such that if \( r_i \) is any other request in that experiment then \( (r_i < r_{\text{max}} \lor r_i \rho' r_{\text{max}}) \). Then the state of each module entered by \( r_{\text{max}} \) at the time \( r_{\text{max}} \) enters that module, is the same as what it would be after the complete execution of all other requests.

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

If \( r_i < r_{\text{max}} \) then by Lemma 3, \( r_{\text{max}} \) can enter a module in \( C_{\text{max}} \) only after \( r_i \) has exited all modules in \( C_{\text{max}} \) for the last time. Furthermore, Restriction 6 insures 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 \( r_{\text{max}} \) 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 process module which initiated the request or cause some other actions which is externally discernable (e.g., write a line to a terminal). Because we must take such actions into account in deciding whether two experiments produce the same results, we shall define a \textit{process pseudo-printer} to record them. Associated with each process will be a pseudo-printer 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 pseudo-printer will include all the particulars of the action. (E.g., address changed and its new contents; line typed at terminal, etc.) The \textit{system state} is defined as the values of all permanent variables in the system and the pseudo-printer output for all processes.

Two experiments are said to be \textit{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 our 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 as this information is not available to a process.
**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 we 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 the complete execution of \( r_i \). 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 \( r_{\text{max}} \) be a request satisfying

\[
(\forall r_i) \ [r_i < r_{\text{max}} \lor r_i \rho r_{\text{max}}].
\]  

Such an \( r_{\text{max}} \) exists due to Lemmas 3 and 5. Let \( X' \) be the asynchronous experiment obtained from \( X \) by deleting \( r_{\text{max}} \). 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 \( r_{\text{max}} \). (Thus \( \hat{x} = \hat{x}' ; r_{\text{max}} \)). 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 \( r_{\text{max}} \) cannot cause any previous request to awaken.

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

Since \( \hat{x}' \) and \( X' \) are equivalent \( r_{\text{max}} \) sees the same state if it runs following \( X' \) or \( \hat{x}' \), thus we conclude that the experiment \( \hat{x}' ; r_{\text{max}} \) is equivalent to \( Y \). \( \hat{x} \) is therefore a synchronous experiment equivalent to \( X \).

IV. CONCLUSION

In this paper a technique has been proposed for simplifying the verification 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 [Hoar74]. Such a mechanism may involve the
use of local variables for calculation of priorities. Also, relaxation of
the restrictions is under consideration.
REFERENCES


