ON RESTRICTIONS TO ENSURE REPRODUCIBLE BEHAVIOR IN CONCURRENT PROGRAMS*

by

A. J. Bernstein

and

F. B. Schneider

TR79-374

A. J. Bernstein
Department of Computer Science
SUNY at Stony Brook
Stony Brook, New York 11790

F. B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

*This research has been supported in part by NSF grants MCS76-04828 and MCS76-22360.

ON RESTRICTIONS TO ENSURE REPRODUCIBLE BEHAVIOR IN CONCURRENT PROGRAMS*

A.J. Bernstein[1]
F.B. Schneider[2]
3/26/79

ABSTRACT

One of the major difficulties encountered when dealing with concurrent programs is that reproducible behavior may not be assumed. As a result, it is difficult to validate and debug such systems. In this paper, structural restrictions are presented that ensure that reproducible behavior will occur in concurrent programs. The application of this to system design is discussed.

Keywords: time dependent behavior, concurrency, synchronization, monitors, Concurrent Pascal.

1. Department of Computer Science, S.U.N.Y. at Stony Brook, Stony Brook, N.Y. 11790.
2. Department of Computer Science, Cornell University, Ithaca, N.Y. 14853.
* This research has been supported in part by NSF grants MCS76-04828 and MCS76-22360.

ON RESTRICTIONS TO ENSURE REPRODUCIBLE BEHAVIOR IN CONCURRENT PROGRAMS

1. Introduction

The ability to reproduce the behavior of a sequential program is generally taken for granted. Since each statement is deterministic and the statements are executed in a well defined sequence, the program is guaranteed to produce the same results each time it is initiated in a particular starting state. The desirability of this property from the point of view of program validation is clear. Reproducibility, however, may not be assumed in the case of concurrent programs--programs that support several sites of activity or processes simultaneously. In such programs each process progresses at an undefined speed. Consequently, although accesses to shared data entities may be serialized by some synchronization mechanism, it may not be possible to specify the order in which these accesses are made. This situation is complicated by the fact that the rate of progress of a process may vary from one execution to the next, since it may depend on such unpredictable and uncontrollable factors as device latency and seek time, cycle stealing, and the vagaries of scheduling algorithms.

Needless to say, programs that do not exhibit reproducible behavior are very difficult to understand and validate. In fact, an exceedingly troublesome class of errors in operating systems, called time dependent errors, are a direct result of this inability to reproduce program behavior. These errors stem from particular execution interleavings of processes that share data. Such interleavings may not have been anticipated by the system designer and may not be reproducible. It is this type of behavior that makes it so much more difficult to create reliable asynchronous systems than to construct sequential programs of similar complexity (e.g., compilers). In this paper, restrictions are defined on the structure of a concurrent program that guarantee that only reproducible behavior can occur.

Some initial work in this area was reported in [ABSS78]. The results described here are developed formally in [S78]. This paper is intended to provide a survey of those results. Note that much of the work regarding consistency in a shared database [EGLT76] can be derived from the results presented here.

2. The Problem of Reproducibility

A system consists of a collection of sequential modules that communicate using a call/return mechanism. It implements a number of functions that can be invoked by user or system processes. (E.g., read a file; send a message). The execution in the system that results when a particular function is invoked by a particular process with specific parameters is referred to as a *request*. Thus, a request is a sequential thread of execution through the system. The behavior of a system is not only dependent on the code executed by requests but on their relative timing as well. Two aspects of the relative timing of request execution are noteworthy. The *external sequencing* of a collection of requests corresponds to the relative times at which requests are actually initiated by the invoking processes. The *internal sequencing* of the requests corresponds to the way execution in modules on behalf of concurrently executing requests is interleaved. Note that whereas external sequencing can be controlled and reproduced, this may not be true of internal sequencing due to factors such as those mentioned above. Thus, the repetition of a given external sequence of requests may not produce the same results even though the system is started in the same state each time.

If concurrent requests execute in disjoint portions of the system, they will produce the same results independent of their internal sequencing. Consequently, the outcome of such execution can always be reproduced by restoring the system to the same initial state and reinvoking the requests. When requests concurrently execute in common modules, the results produced may depend on internal sequencing. Two examples will serve to illustrate types of system structure that give rise to behavior that can not be reproduced in a systematic way. In the following, the monitor construct [B73] [H74] with the automatic signal facility of Kessels [K77] is used to serialize the access of a number of concurrently executing requests to common modules. The results presented, however, are not limited only to systems constructed from monitors. The restrictions that guarantee reproducibility can be formulated for use with other synchronization mechanisms, as well.

2.1. Problem of Multiple Shared Modules

If requests execute in more than one common module (monitor) then nonreproducible behavior may result. For example, in Figure 1 requests r_1 and r_2 execute concurrently in modules m_a and m_b respectively. Assume execution in either m_a or m_b by a request involves making a nested call to both m_1 and m_2, in that order. Clearly m_1 and m_2 must be monitors, and thus simultaneous execution by the two requests in these modules cannot occur. This, however, does not control the order in which the requests enter these modules. For example, r_1 may complete execution in both m_1 and m_2 before r_2 enters either, or r_1 may complete execution in m_1 first, but its entry to m_2 may occur only after r_2 has exited from that module. Clearly, the results produced (i.e., values returned to the calling modules) and the final state of the system may depend on the internal sequencing that has actually taken place. Moreover, since the mutual exclusion associated with m_1 and m_2 is the only factor synchronizing the requests, if the system is restored to its initial state, and the two functions reinvoked in the same external sequence, the interleaving that occurs might be different. Thus, it may be necessary to restart execution many times before a particular outcome is reproduced.

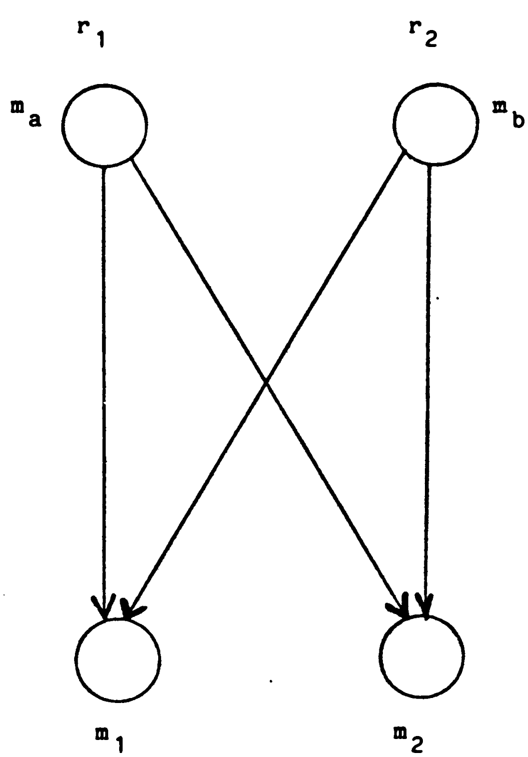

Figure 1

2.2. Problem of Internal Resynchronization

Even if concurrently executing requests share only a single module, nonreproducible behavior may occur. For example, assume requests r_1 and r_2 in Figure 2 become suspended at <u>wait</u> statements in modules m_1 and m_2 respectively. Subsequently, request r_3 causes r_1 and r_2 to be awakened. If upon awakening, both r_1 and r_2 attempt to enter monitor m_3, then a "race" condition exists, as it cannot be predicted which request will actually enter m_3 first. If the order of entry into m_3 affects the computations performed by r_1 and r_2, then it may be difficult to reproduce the system's behavior by processing the same requests in the same external sequence. In this example, modules m_1 and m_2 both contain a <u>wait</u> statement followed by a call to m_3. It is interesting to note that if the order of these two statements in both modules were interchanged, the race condition would be eliminated and the results could be reproduced. Thus, if in the modified system r_1 enters m_3 prior to r_2, the results obtained could be reproduced by restarting the system in the same initial state and reinvoking the requests in the external sequence $r_1\ r_2\ r_3$. In this case it would be necessary to impose the additional constraint on the external sequencing that no request is initiated until all previous requests had either blocked or completed.

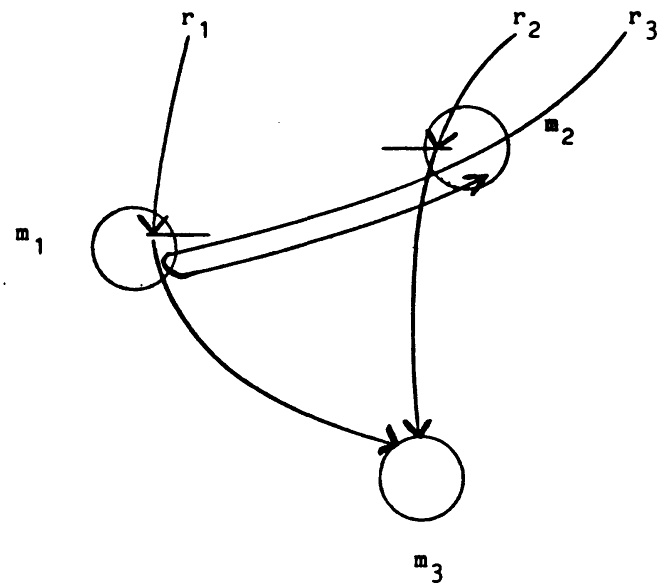

Figure 2

3. Restrictions to Ensure Reproducible Behavior

A system is said to be in a *quiescent state* when all requests that have been initiated by processes have returned. The execution in a system that results from a number of requests will be referred to as an *experiment*. An experiment always starts in a quiescent state and ends when all requests have returned or are suspended in the system. An experiment is *reproducible* if its outcome can always be reproduced by restoring the system to the same starting state and resubmitting the same requests in the same order. A system exhibits *reproducible behavior* if for each experiment there exists a reproducible experiment, involving the same requests, that produces identical results.

A *synchronous experiment* is one in which no request is submitted until all previous requests have either blocked (at a *wait* statement or entry to a monitor) or exited the system. In [S78] it is shown that systems structured in accordance with certain restrictions (to be described in the next section) can only exhibit reproducible behavior. In particular, it is shown that in such systems there exist synchronous experiments that produce results equivalent to those that can be obtained from every other experiment, and that these synchronous experiments are reproducible.

While it is an overstatement to say that only one request is active at a time during a synchronous experiment, it is the case that in a system structured in accordance with the restrictions any concurrency in a synchronous experiment (due to the awakening of previously blocked requests) occurs in disjoint regions of the system.

There are several benefits that accrue from constructing systems that exhibit reproducible behavior. First, the designer of such systems need not be concerned with the interaction of concurrently executing tasks. The designer may think of each request as executing in isolation, with the assurance that the system state seen by the request in the various modules that it enters will be the same as what it could have seen in a synchronous experiment. This simplifies the design task. Secondly, if it can be shown that the set of synchronous experiments produces correct results, then it will be the case that the system functions correctly under any circumstances. Since these experiments are reproducible, their validation is considerably simplified. All errors that occur can be reproduced and tracked down systematically.

It should be noted that, in contrast to other work in this area, the goal of the research described here is to develop conditions sufficient to guarantee reproducibility, not to demonstrate the correctness of a system.

Two restrictions are required to guarantee reproducibility. One restriction deals with the use of *wait* statements, while the second restriction concerns the order in which modules may be invoked by requests. Although the restrictions are sufficient to ensure reproducible behavior in systems, they are not necessary. As a result, systems violating one or the other of the restrictions may still exhibit reproducible behavior. Consequently, certain system structures have been identified where relaxation of the restrictions is permitted. These, too, will be discussed.

3.1. Restriction Concerning Request Trajectories

The first restriction states:

(A) No request may enter a monitor after it has returned from a monitor.

Thus, a request may enter an arbitrary number of monitors, but only by making nested calls. The nonreproducible behavior described in Section 2.1 is due to a violation of this restriction. A consequence of this restriction is that the common modules visited by any pair of requests are all first entered by one request, and then by the other. Roughly speaking, this means that the state of the system, as viewed by any request, is as though all preceeding requests had completed and all succeeding requests had not yet been initiated, even though in actuality a number of requests might be concurrently executing.

Note that this restriction is not a limitation on the functions that can be performed by the system, but rather a restriction on the structure used to implement those functions. For example, the access pattern shown in Figure 1 can be transformed to that shown in Figure 3. In this case, m_3 is a monitor and either m_1 or m_2 may be a monitor (but not both), if necessary. The first restriction is now satisfied. This structure has the disadvantage of reducing the amount of parallelism in the system. It is no longer possible for requests that originate in m_a and m_b to execute in m_1 and m_2 concurrently. However, the added parallelism afforded by the structure in Figure 1 may cause nonreproducible behavior, and consequently this reduction of potential parallelism is deemed advantageous.

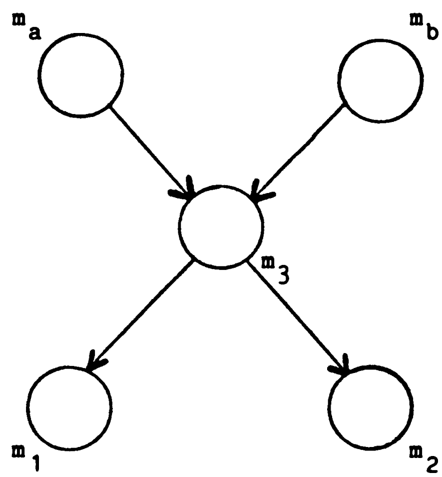

Figure 3

3.2. Restriction Concerning Wait Statements

A distinction can be made between a *wait* statement that is the first executable statement of a monitor procedure, and one that is not. The former are referred to as i-*waits* (for *i*nitial *wait*); the latter as m-*waits* (for *m*iddle *wait*). The second restriction concerns only m-*waits*, and states:

(B) A request may not call any monitors after being suspended at an m-*wait*.

Note that the nonreproducible behavior illustrated in Section 2.2 is a consequence of a violation of this restriction.

It is not difficult to obtain an intuitive grasp of why this restriction does not apply to i-*waits*. Since a process suspended at an i-*wait* statement has not had the opportunity to either modify the state of the monitor or store information in local variables, it is similar to a process that is suspended at monitor entry due to mutual exclusion. In both cases the actual execution within the monitor is being delayed. Since a request may perform a sequence of nested monitor calls without violating the first restriction, such calls should be allowed after i-_wait_s.

The _wait_ statement is usually employed to delay a request because the state of a monitor is not conducive to its continued execution. It would, therefore, seem that the i-_wait_ should suffice in most situations and thus, this restriction should not constrain the system designer excessively. This is confirmed by examining the examples discussed in [H74]. It is often necessary, however, to do computations for the purpose of scheduling prior to executing a _wait_ statement. For example, a disk scheduling monitor may wish to order waiting processes in accordance with the disk track they intend to access. These computations neither change the monitor permanent variables nor retain information about the monitor state. They are used solely to order the suspended processes. If these are the only statements prior to a _wait_ statement, they should be permitted without having to treat the _wait_ statement as an m-_wait_. The reasoning here is identical to the justification for exempting i-_wait_s from restriction (B), since only the order in which processes are awakened is involved.

A generalization of the priority _wait_ mechanism of Hoare [H74] has been defined which can be used to bind an arbitrary computation to a _wait_ statement [ABS77] [SB79] for this purpose. Thus, a _wait_ statement preceded only by a priority computation is still considered an i-_wait_. An example is shown in Figure 4. A priority function, which may only modify its own local variables, is bound to a condition by a _use_ clause. The function computes an integer valued priority based on parameters passed to it at the time the _wait_ statement is executed if the associated condition is found to be false. This priority is used to control the order in which processes suspended on the condition are awakened. Figure 4 is a single resource monitor [H74] with a priority function included to ensure that access to the resource is granted in ascending order by "prty", a parameter passed to the monitor.

```
type single_resource = monitor;
    var inuse : boolean;
        free: condition {not inuse} use prio;
    function prio (p: integer): integer;
        begin
            prio := p
        end;
    procedure entry allocate (prty: integer);
        begin
            free . wait (prty);
            inuse := true
        end;
    procedure entry deallocate;
        begin
            inuse := false
        end;
    begin
        inuse := false
    end;
```

Figure 4

It should be noted that despite this generalization, there exist situations where an m-*wait* is required. A monitor serving as a rendezvous point for asynchronous requests is an example of such a situation, and restriction (B) would apply in that case.

3.3. Relaxation of the Restrictions--Resource Schedulers

Attempts to program schedulers have motivated two system structuring mechanisms that preserve reproducible behavior while at the same time allowing relaxation of restriction (A). A scheduling scheme may follow the pattern of Figure 1 (see [B75], for example). In this case, m_1 is a module that schedules access to module m_2. Modules m_a and m_b correspond to potential users of m_2 and as such, must have access rights to both m_1 and m_2. Thus, m_1 and m_2 must be monitors. Only cases in which the scheduler allows one request at a time to access the module being scheduled will be considered. As a result, that module will not be subject to concurrent access. A module to which access is so regulated by a scheduler will be referred to as a _resource_, in order to distinguish it from other modules in the system.

A request first makes an _allocate_ call to m_1 to gain access to the resource. Since the resource is shared, the request may be suspended in m_1 until the resource is free. At that time, the return from m_1 to m_a (or m_b) occurs, and the request calls m_2 to use the resource. Upon completion, the request makes a _deallocate_ call m_1 to indicate that the resource is once again free.

Although this protocol violates restriction (A), note that no information need be returned by m_1 to its callers (i.e., all parameters passed to m_1 can be value parameters). If this is the case then there is no way that a request from m_a, following the protocol, can detect the interleaving that might occur as a result of a similar and concurrent pattern of calls originating in m_b. This fact can be exploited to relax restriction (A) without sacrificing reproducible behavior.

A _scheduler_ is defined to be a monitor that controls access to a resource through allocate and deallocate procedures and exhibits the following properties:

(i) All parameters passed to a scheduler are passed by value.

(ii) The scheduler grants at most one request access to the resource at any time.

(iii) The deallocate procedure contains no wait statement.

(iv) The allocate procedure delays a request only if the resource is busy.

(v) Schedulers do not possess access rights to monitors.

Using this description of a scheduler it can be shown that restriction (A) can be replaced by the following pair of weaker conditions. The term "monitor" is not meant to refer to either schedulers or the resources they schedule.

(A1') No request may call a monitor after calling a deallocate procedure (of a scheduler) or after it has returned from a call to a monitor.

(A2') A request may not call an allocate procedure (of a scheduler) after calling a deallocate procedure (of a scheduler) or after exiting from a monitor.

Restriction (B) is replaced by

(B') A request may not call any monitor or any allocate procedures (of schedulers) after being suspended at an m-wait.

Under these restrictions the calling sequence described with respect to Figure 1 can be allowed, without sacrificing reproducibility, provided m_1 is a scheduler.

An additional restriction is needed for systems in which requests may require access to several resources, each with its own scheduler. This deals with the order in which the schedulers may be invoked. It is required to prevent certain forms of deadlock involving the various resources. The restriction is:

(C) Requests must acquire access to resources in a fixed (hierarchical) order [H68].

In [B75], a simple spooling system was presented to illustrate the use of Concurrent Pascal. In that system, a resource scheduling strategy similar to that of Figure 1 is used to control access to a disk and to a console. The modules performing the scheduling satisfy properties (i) through (v). However, restriction (A2') is violated and, not surprisingly, the system does not exhibit reproducible behavior. In particular, in the event of a disk I/O error, access to the disk resource is relinquished (by making a deallocate call to the disk scheduler) and then access to the console is acquired (by making an allocate call to the console scheduler) so that an error message can be written to the console. A consequence of this is that the order in which messages appear on the console is not reproducible and may be misleading, since it may not correspond to the order in which the disk operations were attempted. This would make analyzing disk errors in the system difficult, as one can not determine the actual order in which the I/O operations were attempted by reading the console output. (Note: No assertion was made in [B75] that the system would exhibit reproducible behavior, nor that console messages would reflect the actual order in which I/O was attempted.) If the disk were deallocated after the console was accessed, none of the restrictions would be violated and the system would exhibit reproducible behavior. In addition, the order in which messages appear on the console would reflect the order of attempted disk accesses.

The notion of a scheduler has been elaborated to a module, called a manager [SKB77], that dynamically allocates access rights (capabilities) for a number of identical resources. For example, a system having a limited number of buffers might wish to allocate them dynamically to processes. Each buffer could be implemented as a module that supports read and write operations. If buffers are always initialized prior to allocation, then all instances will behave identically, and a request need not know the identity of the particular instance that has been allocated to it. The manager takes advantage of this by responding to requests for allocation with sealed capabilities. Thus, no information flows from the manager back to the caller--a similar situation to that described for schedulers. Managers are characterized by modifying (i) to allow a sealed capability to be returned, and adding the following property:

(vi) All resources allocated by a manager appear identical to requesting modules.

Furthermore, property (iv) can be relaxed somewhat to admit very general resource allocation schemes (e.g., the bankers algorithm [B73]). The more general form of that property is discussed in [S78]. Note that whereas a manager always allocates initialized resources, a scheduler does not. Using this description of a manager it is shown in [S78] that restrictions (A1'), (A2') and (B') can be weakened so that term "monitor" is not meant to refer to schedulers, managers or resources.

3.4. Relaxation of the Restrictions--Monitor Entry

In situations where each call to a resource must be individually scheduled (as in a disk), the system structure of Figure 1 has disadvantages. In particular, the protocol outlined above is undesirable since it reveals to higher levels of the system (e.g., m_a, m_b) the functions of scheduling and resource use as separate entities and requires separate invocation of each. Furthermore, in systems where access rights are determined statically (e.g., Concurrent Pascal [B75]), it is very difficult to guarantee at compile time that the protocol is observed. A more desirable arrangement would be to provide a single call that invoked both the scheduling and use functions, and returned to the higher level on completion. Although Figure 5 exhibits this structure,

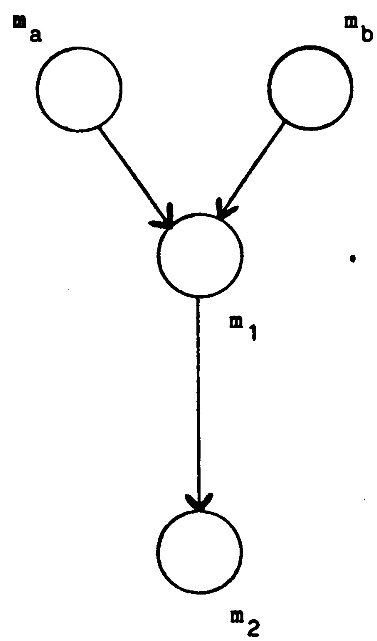

Figure 5

(m_1 schedules access to m_2 for m_a and m_b), it is unacceptable because entry to the scheduling module is prevented (by the mutual exclusion at m_1) if the resource is in use. Thus, no real scheduling can take place.

To solve this problem a mechanism has been proposed to allow specification of the order in which processes suspended at monitor entry are allowed to proceed [SB78] [SB79]. The mechanism is a generalization of the priority <u>wait</u> described earlier and allows a priority function to be bound to each monitor entry procedure. This permits a simple solution to the scheduling problem. Furthermore, for the same reason that priority function computations were permitted to precede i-_waits_, this scheduling mechanism does not affect the reproducibility of system behavior. Processes calling a monitor at a time when another process is actively executing within are made to wait and are ordered according to a priority computed by the priority function associated with the entry procedure that they have called. When activity within the monitor ceases, the waiting process with the highest priority is allowed to enter. An illustration of the use of a priority function at monitor entry appears in Figure 6. A monitor is defined there that guards access to a resource. Accesses to that resource are granted in priority order, based on a monitor entry parameter. More complex scheduling disciplines can be enforced as well (e.g., the elevator disk head scheduling algorithm [SB79]).

```
type prio_access = monitor;
    function prio (p: integer): integer;
        begin
            prio := p
        end;
    procedure entry access (p: integer) use prio(p);
        begin
            call resource
        end;
    begin
    end;
```

Figure 6

4. Discussion

It is the programmer's job to construct a module so that every routine leaves it in a state that satisfies the module invariant [H74]. There often exist relations, however, that link the states of different modules. It is the responsibility of the programmer to guarantee that the interleaved execution of concurrent requests does not create a situation where a violation of such a relation could be observed. For example, in a file system the directory for a disk may be managed by one module, while the available free space by another. Clearly, the same track should never appear listed in both modules, as a track may be either allocated to a file or free. This constraint on the valid system states may be specified by an invariant relation that links the states of more than one module. Such a relation will be called an *intermodule invariant*. For obvious reasons, such invariants can be assumed to be true only while there are no requests executing in the system.

A *system invariant* is a relation that characterizes all the valid system states. The system invariant is not necessarily the conjunction of all the module invariants of the components of the system because restrictions imposed by intermodule invariants must also be taken into account. A consequence of the result developed above is that in a system in which the system invariant is satisfied by the state that results from every synchronous experiment, only states that will satisfy that invariant oan result from concurrent execution. Thus, no state that violates an invariant can be seen by a single request.

It is the system designer's job to try to decompose a system into modules in such a way that there are no intermodule invariants. A great deal of effort has been devoted to developing design methods that are supposed to yield such a decomposition [P72] [WF77]. Unfortunately, the state of the art is not such that this is always possible, nor is it always desirable. Consequently our view of valid system states as being constrained by intermodule invariants seems realistic.

The nature of the proposed structural restrictions will govern the applicability of the theory to real systems. The restriction regarding wait statements does not appear to be severe. None of the examples studied (including all those in [H74]) violate this restriction. The other restriction, although more severe, can be relaxed in many situations, as has been illustrated. Furthermore, there is evidence that a weaker form of this restriction can be formulated in terms of intermodule invariants. Note that if all synchronous experiments leave intermodule invariants true, then if two modules are linked by such an invariant there must exist a request that visits both of them. Otherwise it would be possible to run a synchronous experiment involving 1 request that did not leave the system with all intermodule invariants satisfied. The requirement that when a request visits more than one monitor it does so by making nested calls ensures that no request may visit a monitor while it is in a state that does not satisfy an intermodule invariant--whether or not such an invariant exists! Thus, it is expected that the first restriction can be weakened to apply only to modules whose states actually are linked by intermodule invariant relations.

Unfortunately, the proposed restrictions are not, in general, compile time checkable. However, adherence to the restrictions can be checked on a module by module basis. The simplicity of the restrictions and the ease of validating compliance with them makes their use by practitioners possible. This contrasts with other work aimed at the construction of reproducible systems [Si78].

5. Conclusions

Structural retrictions have been presented that ensure reproducible behavior in concurrent programs. Thus, any result produced by such restricted systems can be reproduced in a systematic way. Consequently, understanding and validating such programs is simplified. In addition, proofs of such programs are simpler as the number of interleavings that must be considered is reduced. This makes it easier to establish the interference free property required by Owicki and Gries [OG76].

6. Acknowledgment

The authors would like to acknowledge contributions made at an earlier stage of this research by Professors E. Akkoyunlu and A. Silberschatz.

[S78]
Schneider, F.B., "Structure of Concurrent Programs Exhibiting Reproducible Behavior," Ph.D. Thesis, S.U.N.Y. at Stony Brook, August 1978.

[SB78]
Schneider, F.B. and A.J. Bernstein, "Scheduling in Concurrent Pascal," Operating Systems Review 12, April 1978.

[SB79]
Schneider, F.B. and A.J. Bernstein, "Mechanisms to Allow Specification of Scheduling Policies," submitted for publication.

[Si78]
Silberschatz, A., "Serializability in Multi-Level Monitor Environments," Proc. 1978 ACM Annual Conference.

[SKB77]
Silberschatz, A., R.B. Kieburtz and A.J. Bernstein, "Extending Concurrent Pascal to Allow Dynamic Resource Management," IEEE Transactions on Software Engineering SE-3,3, pp. 210-217.

[WF77]
Wasserman, A. and P. Freeman, Software Design Techniques, IEEE Computer Society, Long Beach, CA, 1977.

REFERENCES

[ABS77]
Akkoyunlu, E.A., A.J. Bernstein, and F.B. Schneider, "Medium Term Scheduling and Equivalence of Synchronous and Asynchronous Operation," S.U.N.Y. at Stony Brook, Computer Science Department Technical Report TR-72 (June 1977).

[ABSS78]
Akkoyunlu, E.A., A.J. Bernstein, F.B. Schneider and A. Silberschatz, "Conditions for the Equivalence of Synchronous and Asynchronous Systems," IEEE Transactions on Software Engineering, SE-4,6, pp. 507-516.

[B73]
Brinch Hansen, P., Operating System Principles, Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[B75]
Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, SE-1,2, pp. 199-206.

[EGLT76]
Eswarran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11, pp. 624-633.

[H68]
Havender, J.W., "Avoiding Deadlock in Multi-Tasking Systems," IBM Systems Journal 7, 2, pp. 74-84.

[H74]
Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM 17, 10, pp. 549-557.

[K77]
Kessels, J.L.W., "An Alternative to Event Queues for Synchronization in Monitors," CACM 20, 7, pp. 500-503.

[OG76]
Owicki, S. and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica 6, pp. 319-340, (1976).

[P72]
Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, 12, pp. 1053-1058.