On Language Restrictions to Ensure Deterministic Behavior in Concurrent Systems

A. J. Bernstein
F. B. Schneider

Dept. of Computer Science
S.U.N.Y. at Stony Brook
Stony Brook, New York U.S.A.

Abstract:
Software that supports concurrent operation of asynchronous processes often produces results which are timing dependent. As a result it is difficult to validate and debug such systems. A set of language restrictions is presented which ensures that timing dependent behavior can not occur. Concurrency is not excessively limited by these restrictions, however. The application of this to system design is discussed.

*This research was supported in part by NSF grant MCS 76-04828.

The profusion of software systems that do not function as they should is alarming. Research to control errors in software ranges from theoretical to practical approaches. Verification of programs by axiomatics [Wic76], [Hoar69] and other formal models [Denn72], although presently not applicable to large systems, shows promise of becoming one means to eliminate errors. In the meantime, the development and widespread adoption of high level languages for systems programming, structured programming techniques [Dijk72], modularization [Parn74] [Lisk74] [Wulf76], and hierarchical system construction [Dijk68] have been successfully employed to control errors in the construction of software.

The most difficult software systems to validate or verify are those which involve concurrent processes. Operating systems are examples of such systems. They support a number of independent processes which share data, perhaps using some synchronization mechanism to support cooperation. It is often the case that such systems exhibit timing dependent or non-deterministic behavior.

For example, consider the system access graph of Figure 1. In this graph a node represents a module and an arc from one node to another indicates that the former module may make a call upon the latter [Brin75]. Assume the algorithms implemented in $m_1$ and $m_2$ involve successive calls to $m_1$ and $m_2$. If processes $r_1$ and $r_2$ are simultaneously executing in $m_1$ and $m_2$, timing dependent behavior may ensue. The following sequence of events illustrates this:

1. $r_1$ enters $m_1$
2. $r_2$ enters $m_2$
3. $r_1$ enters $m_a$
4. $r_1$ returns to $m_a$
5. $r_2$ enters $m_1$
6. $r_2$ returns to $m_1$
7. $r_2$ enters $m_2$
8. $r_1$ enters $m_2$
9. $r_1$ returns to $m_2$

Notice that process $r_1$ sees the state of module $m_2$ before $r_2$ has entered it but sees the state of $m_2$ after $r_2$ has completed execution in that module. This behavior is the result of a particular interleaving of the execution of concurrently running processes.

![Figure 1](image-url)
It may not be possible to obtain the same results by running $r_1$ and $r_2$ in a serial fashion nor can we be sure that the same interleaving will occur if we attempt to reproduce the sequence of events by initiating $r_1$ and $r_2$ at the same relative times. This is because the rate of progress of a process in the system may depend upon such unpredictable factors as latency, seek times, and the vagaries of a scheduling algorithm. As a result it may be very difficult to reproduce a particular sequence of events and thus the results produced may be timing dependent. The system designer should have anticipated all possible interleavings of execution in modules. Nevertheless, systems which exhibit this type of behavior are particularly difficult to validate.

It is the goal of this research to define a set of language restrictions to ensure deterministic operation of asynchronous systems. These restrictions impose sufficient structure on the system so that timing dependent (non-reproducible) behavior cannot occur. Unfortunately not all of the restrictions are compile time checkable, though the run time checks incur minimal overhead. In addition, we believe the restrictions leave enough flexibility so that the application of this theory to real system implementation is feasible.

A system is viewed as a passive entity: a hierarchically organized collection of sequential modules. These modules interact with each other via a call/return mechanism. In addition, we postulate the existence of active entities, processes, which initiate requests. A request is the execution which results in the system when a process makes a call to the system. Each process is assumed to be sequential and executes asynchronously with other processes. Thus no assumptions are made about the relative speeds of the processes, only that each will make finite progress.

A monitor [Brin73] [Hoar74] is an extension of the abstract data type concept [Lisk74]. The latter was first developed in the language Simula [Dah68] and called a class. A class is a module which encapsulates a set of variables and all the routines which access them. Access to the variables is not permitted outside the module. They are referred to as permanent variables since their values are retained between calls (as with Algol own variables). Thus, a class provides to other modules of the system a high level data object whose implementation details are hidden and which can be manipulated by the high level operations supported by the module's procedures.

A monitor extends these ideas so that they can be used in an asynchronous environment.

It is a module that consists of a number of entry procedures, local procedures and permanent variables. The only way to execute in a 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 permits 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 relinquish control of a monitor in the event that the state is not conducive to continued execution. We assume the conditional wait facility proposed by Kessels [Kess77] (though others may be used). When a process executes a wait statement, a boolean expression associated with the statement is evaluated. If the value produced is false, the process is suspended at that statement and the mutual exclusion associated with the monitor is released. The process is subsequently reactivated if there is no process active in a monitor procedure and the boolean condition that caused the process to be suspended is no longer false. An example of a monitor is shown in Figure 2.

```plaintext
type single_resource = monitor;
var inuse : boolean;
ok : condition [not inuse];
procedure entry acquire ;
begin
  ok.wait ;
inuse := true
end ;

procedure entry release ;
begin
  inuse := false
begin
  inuse := false
end

Figure 2
```

We will describe the results of the research in terms of the language Concurrent Pascal [Brin75], a systems implementation language based on Pascal [Wirt71]. Concurrent Pascal implements the monitor construct described above as well as the class construct. Several systems have been implemented in this language [Brin76] [Graf77].

The execution in the system that results from a number of requests will be called an experiment. If the system is constrained so that no request is submitted until the previous request has either been blocked at a wait statement or mutual exclusion, or exited the system then we shall call the execution that results a synchronous experiment. Alternately, the situation in which more than
ON LANGUAGE RESTRICTIONS

539

one request is processed at a time will be called an asynchronous experiment.

Note that a certain limited form of concurrent activity can occur in a synchronous experiment. This would happen if a request awakens a previous request which was suspended in the system. Thus a synchronous experiment may exhibit timing dependent behavior. In this paper we will describe certain restrictions which can be imposed on the way a system is built. These restrictions guarantee that corresponding to every experiment there exists a synchronous experiment which, starting from the same initial state and involving the same requests, produces identical results [Sch78a]. Furthermore, the corresponding synchronous experiment is entirely reproducible - it exhibits no timing dependent behavior. The implication of this is that if it can be shown that the set of corresponding synchronous experiments produce correct results then it will be the case that the system functions correctly under any circumstances. Since these experiments are not timing dependent their validation is considerably simplified. There are several benefits which accrue from such an equivalence result. First, the designer of systems that adhere to these restrictions 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 the equivalent synchronous experiment. This simplifies the design task. Similarly, such systems will exhibit no timing dependent errors, a particularly difficult type of error condition to isolate and reproduce which frequently asynchronous systems. Consequently, the difficulty of validating (and debugging if necessary) such systems is considerably reduced as one need only be concerned with the set of corresponding synchronous experiments. Lastly, proofs of systems that exhibit the equivalence property are simplified. Normally a proof of a concurrent program shows that every possible interleaving of execution in all system modules yields acceptable results. It can be shown that as a consequence of the restrictions the number of interleavings that can occur is greatly reduced.

The language restrictions necessary to guarantee the equivalence between an arbitrary experiment and some reproducible synchronous experiment will now be presented. As the formal development of the results is rather laborious, we will restrict ourselves to an informal discussion of the theory. A formal development can be found in [Akko77a] [Akko77b] [Sch78a].

One restriction deals with the use of the wait statement and specifies that it may appear as the first and/or last statement of a monitor procedure. Since it is often necessary to do some computation prior to the wait statement for purposes of scheduling, a generalization of the Hoare priority wait scheme [Hoar74] has been defined called the generalized condition [Akko77b]. This facilitates certain computations associated with waiting even though the wait statement syntactically appears as the first statement of a monitor procedure. Wait statements can also appear in the middle of monitor procedures, although subject to the restriction that no monitor is called by a request after such a wait statement is executed.

A second restriction specifies that no request attempt to enter a monitor after it has exited from a monitor. The timing dependent operation illustrated in Figure 1 is a consequence of a violation of this restriction (monitor m₁ is entered after return from monitor m₂). From this restriction it follows that the common modules visited by any pair of requests are all visited first by one request and then by the other. This means that the state of the system as viewed by any request is as though all preceding requests had completed, and all succeeding requests had not yet been initiated, even though these requests might be executing concurrently.

This restriction is not a limitation on the functions which can be performed by the system but rather a restriction on the structure used to implement those functions. Thus a request executing as described in Figure 1 can be divided into two separate requests, one which executes in m₁ and the second in m₂. Each of the new requests now satisfies the restriction. This has the effect of exhibiting in synchronous experiments results which could only have been produced by interleaved requests in asynchronous experiments in the original design. Thus the equivalence result is achieved. This transformation unfortunately complicates the user interface, i.e., the user must make two system calls instead of one. A second transformation involves changing the topology of the access graph as illustrated in Figure 1. In this case m₁ is a monitor and either m₁ or m₂ may be a monitor (but not both) if necessary. Once again the restriction is 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 m₁ and m₂ to execute in m₁ and m₂ concurrently. However, the added parallelism afforded by the structure in Figure 1 is exactly what may cause the undesired timing
dependent behavior, and consequently we contend this reduction of potential parallelism is advantageous.

The resource scheduling algorithm described above has an important deficiency when implemented in a language (e.g., Concurrent Pascal) which statically allocates access rights for inter-module calls. Since $m_1$ must call $m_2$, it must be granted an access right for that purpose at initialization time. As a result it is very difficult to guarantee at compile time that $m_1$ is called prior to each call to $m_2$. Indeed, $m_1$ may call $m_2$ without calling $m_1$ first, causing the system to function in an unpredictable way. If dynamic allocation is permitted then $m_1$ can return to $m_0$ an access right (capability) for $m_2$ at the time the calling process is scheduled and the right can be returned when use of the resource is completed. In this way correct execution of the protocol can be assured. This notion has been extended in the manager concept [Silb77] to a module which schedules access to a number of identical resources. Although in this case the manager returns information to its caller, the information is an access right and can be sealed. As a result, the calling module receives no information which might cause it to produce results which depend upon whether or not an interleaving has occurred. Thus we can conclude that if in Figure 1, $m_1$ is a manager which functions correctly, equivalence will be preserved.

Although dynamic allocation of access rights can be used as described above to provide some assurance that the protocol for resource use will be applied correctly, it does not come to grips with a second problem associated with this scheme. The protocol is undesirable since it reveals to higher levels of the system (i.e., $m_1$, $m_2$) the function of scheduling and reserved use as separate entities and requires separate invocation of each. A more desirable arrangement would be to provide a single call which invoked both the scheduling and use functions and returned to the higher level on completion. Although Figure 3 exhibits this type of structure (with $m_1$, a scheduling monitor and $m_2$, the module which controls the resource) it is unacceptable because entry to the scheduling module is prevented (by the mutual exclusion at $m_1$) if the resource is in use and thus no real scheduling can take place. To solve this problem a priority function may be bound to the monitor entry operation [Schm78]. This permits requests to be admitted to the a module in an order which is computed by this priority function. The scheduling discipline to be imposed is embodied in the priority computation. Scheduling can be done by incorporating the priority function directly in the module controlling the resource. This permits a simple solution to the scheduling problem. Further, the equivalence between synchronous and asynchronous operation is preserved by the mechanism.
The question of the applicability of this theory to the design and implementation of real systems remains open. Presently we are studying this problem with encouraging results. In addition, the generalization of the results to message-based systems is also under study.

References


[Akko77b] Akkoyunlu, E. A., A. J. Bernstein, F. B. Schneider, "Medium Term Scheduling and Equivalence of Synchronous and Asynchronous Iteration", TR-72, Dept. of Computer Science, SUNY at Stony Brook (June 1977)


