Specification and Proof Techniques for Serializers

CARL E. HEWITT AND RUSSELL R. ATKINSON

Abstract—This paper presents a specification language, implementation mechanism, and proof techniques for problems involving the arbitration of concurrent requests to shared protected resources whose integrity must be preserved. This mechanism is the *serializer*, which may be described as a kind of protection mechanism, in that it prevents improper orders of access to a protected resource. Serializers are a more structured form of the monitor mechanism of Brinch Hansen and Hoare.

Serializers attempt to systematize and abstract desirable features of synchronization control structure into a coherent language construct. Serializers have better synchronization modularity than monitors in several respects. Monitors synchronize requests by providing a pair of operations for each request type (examples are STARTREAD/ENDREAD and STARTWRITE/ENDWRITE for the readers–writers problems). Such a pair of operations must be used in a certain order for the synchronization to work properly, yet nothing in the monitor construct enforces this use. Serializers incorporate this structural aspect of synchronization in a unified mechanism to guarantee proper check-in and check-out by encasing the resource inside the serializer. Thus a serializer for a readers–writers problem provides only two entry points for external users: READ(directions) and WRITE(directions) where the directions are passed to the resource.

In scheduling access to a protected resource, it is often necessary that a process wait for a certain condition before continuing execution. Monitors require that a process waiting will remain dormant forever, unless another process explicitly signals the dormant process that it should continue. Serializers improve the modularity of synchronization by providing that the condition for resuming execution must be explicitly stated when a process waits, making it unnecessary for processes to signal other processes. That is, the serializer decides for each process the conditions required for the further execution of the process.

The behavior of a serializer is defined axiomatically in terms of the precedes relation among events using the actor message-passing model of computation [11], [10], [16], [1]. Different versions of the "readers-writers" problems are used in this paper to illustrate how the structure of a serializer corresponds in a natural way to the structure of the specification of synchronization problems.

In this paper we present specification and proof techniques using partial orders on computational events for dealing with problems involving *fairness*, *starvation*, and *guaranteed concurrency*. Our techniques represent a significant advance over previously developed techniques using global states.

Index Terms—Monitor, parallelism, program proving, readers–writers, serializer, starvation, verification.

Manuscript received December 1, 1976; revised June 12, 1978. A very preliminary version of this paper was presented at the ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication in March 1975.

The authors are with the Artificial Intelligence Laboratory and the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

I. Scheduling Access to Resources

We SEE a need for the development of language constructs that are at least partially chosen for their provability. A language feature providing synchronization should be designed to provide usable axioms about the possible orders of events in a program. The language feature should guarantee that conditions needed to prove properties of programs are explicit in the axioms for the language feature.

Serializers have been designed to facilitate the proof that schedulers implemented using them satisfy their specifications. The specifications of a protected resource typically involve stating both integrity and scheduling constraints. An integrity specification typically takes the form of a consistency constraint. A typical example of an integrity specification might be that the position and velocity of an airplane must be recorded for the same instant of time. A scheduling specification typically takes the form of a constraint on the time order of certain events. A typical example of a scheduling specification is that if two requests to write in a data base are received in a certain order then the first request received will be honored before the other. We would like to be able to demonstrate how implementing protected resources using serializers makes it easier to prove that they satisfy their specifications. In particular, we would like to develop techniques for proving that schedulers implemented using serializers guarantee a reply to each request received. Guaranteeing that a reply will be sent for every request received is a stronger and more useful property than merely being free of deadly embrace, which is the scheduling specification most extensively treated in the literature on synchronization.

II. Serializers

A. Concept of a Serializer

In this section we will describe an abstract mechanism called a *serializer* for guaranteeing the integrity of a protected resource. The mechanism is an abstraction and encapsulation of the method commonly used in operating systems. A detailed analysis of the facilities needed will be used to motivate our design decisions.

A serializer bears an analogy to the front desk of a hospital in that only one person can check in or out at a time. The front desk of a hospital serves to schedule the entrance and exit of people in the hospital. Entering or leaving the hospital is impossible without checking through the front desk. Various waiting rooms are maintained for people who are waiting.

In addition records are maintained of where people are within the hospital.

Serializers are modular in the sense that they can be constructed to *encase* the resource to be protected in such a way that it can only be accessed by passing through the serializer. *A serializer should be constructed to surround the protected resources in such a way that it is impossible to accidently avoid passing through it when using the protected resources.* We shall avoid in this paper the issues involved with exactly how one guarantees that a serializer has sole possession of a resource, or even if cooperating serializers might share access to a resource. The reader may assume that every serializer we deal with in this paper has sole access to the resources it encases.

The diagram below shows a simple example of serializer scheduling access to a protected resource 𝔭:

```
         ENTRY
            |
+-----------|---------------------------------------+
|...........V.......................................|
|...................................................|
|.....GUARANTEE.....................................|
|...........|.......................................|
|..+--------|-------+...........JOIN-CROWD..........|
|..|        V       |................|..............|
|..|                |......+---------|--------+.....|
|..|  ESTABLISHED   |......|         V        |.....|
|..|        |       |......|                  |.....|
|..+--------|-------+......| +--------------+ |.....|
|...........|..............| |              | |.....|
|...........V..............| | RESOURCE p   | |.....|
|..........................| |              | |.....|
|..........................| +--------------+ |.....|
|..........................|                  |.....|
|..........................|   LEAVE-CROWD    |.....|
|..........................|         |        |.....|
|..........................+---------|--------+.....|
|.........EXIT.......................|..............|
|...........|........................V..............|
|...........|.......................................|
+-----------|---------------------------------------+
            |
            V
```

Diagram of Serializer Encasing a Protected Resource

Each arrow in the above diagram is labeled with the kind of computational event it represents. All of the events represented in the diagram are serialized in time. Each event represents an occurrence in which a process either (re)gains or releases possession of the serializer. The fact that a process has possession of the serializer is represented by executing in the shaded region of the diagram. At most one process can have possession at any one time.

A *typical* simple sequence of events occurring in the use of a protected resource 𝔭 begins with a SERIALIZER-REQUEST event in which the serializer receives a message M which is intended for the protected resource 𝔭. The request must eventually result in an ENTRY event which gains possession of the serializer. A GUARANTEE event will cause a process to wait until some condition is true before proceeding. Such a request releases possession of the serializer. If execution of a process continues after a GUARANTEE event then the next event will be called an ESTABLISHED event because the condition is guaranteed to be established to be true at the time. Thus each ESTABLISHED event regains possession of the serializer at a point in time when the condition is guaranteed to be true. When the proper condition for using a protected resource has been established, then possession of the serializer can be released by a JOIN-CROWD event which records that there is another process in a crowd which is an internal data structure of the serializer that keeps track of which processes are using the resource. Next the message M is delivered to the protected resource 𝔭 in a RESOURCE-REQUEST event. Eventually the protected resource 𝔭 may produce a reply R to the request which will be called a RESOURCE-REPLY event. The RESOURCE-REPLY will eventually result in a LEAVE-CROWD event which regains possession of the serializer and records that the process is no longer in the crowd using 𝔭. After this the process releases possession with an EXIT event, which causes a SERIALIZER-REPLY event in which the message R is sent as the reply to the original SERIALIZER-REQUEST event.

Serializers derive their name from the fact that all of the events that gain and release possession of the serializer are totally ordered (serial) in time. We assume that every serializer is written such that an event gaining possession is always followed by one releasing possession (usually this is trivial to demonstrate). In the above diagram the interior of the serializer has two "holes," in which a process temporarily releases possession of the serializer. The purpose of a hole entered by a GUARANTEE event is to release possession while a process is waiting for some condition to be established so that it can proceed. This kind of hole is called a "**waiting room.**" The purpose of a hole entered by a JOIN-CROWD event is to allow parallelism in the use of protected resources by releasing possession of the serializer so that other processes can gain possession. This kind of hole is called a "**resource room.**" There may be any number of holes of either variety.

The events of a serializer fall into two disjoint categories that are totally ordered in time: those which GAIN-POSSESSION and those which RELEASE-POSSESSION of the serializer. Each event in the former category subsequently results in an event of the latter category. Furthermore after a GAIN-POSSESSION event has occurred, then another such event will not occur before a RELEASE-POSSESSION event has occurred. To understand the behavior of serializers, one must understand the ways that possession of a serializer is gained and released.

There are four ways to gain possession of a serializer:

1) an ENTRY event, which gains possession as a result of a SERIALIZER-REQUEST event;

2) an ESTABLISHED event, which regains possession as a result of a GUARANTEE event with the condition established to be true;

3) a LEAVE-CROWD event, which regains possession as a result of a RESOURCE-REPLY event from a protected resource;

4) a TIME-OUT event, which regains possession as a result of a process waiting for a condition for a longer period of time than specified when it entered the waiting room.

There are three ways to release possession of a serializer:

1) a GUARANTEE event, which occurs in order to guarantee that some condition is true before continuing execution;

2) a JOIN-CROWD event, which records that a process is using a protected resource;

3) an EXIT event, which causes a reply to the original SERIALIZER-REQUEST event.

There are two reasons that a process using a serializer might have to wait. The first is that it might be necessary for a process in a waiting room to wait for some condition to be established. The second reason for waiting is that a process must wait for possession after a SERIALIZER-REQUEST or a RESOURCE-REPLY event because some other process has possession. Processes waiting for a condition to be established are given absolute priority over other processses waiting for possession.

Each time possession of a serializer is released, processes waiting for a condition to be established are given the opportunity to continue execution. This property of serializers simplifies proofs that a scheduler guarantees replies to requests received and increases the responsiveness of schedulers by allowing processes waiting for a condition to proceed as soon as possible. Roughly speaking, if there are any processes waiting for a condition that are "ready to go" when possession of a serializer is released then the next event to gain possession of the serializer must be an ESTABLISHED event which gives one of those processes possession.

When servicing processes waiting for possession after a SERIALIZER-REQUEST or RESOURCE-REPLY, several kinds of waiting disciplines are possible to determine which process gets possession of the serializer next. The only property that is used in this paper is that every SERIALIZER-REQUEST and RESOURCE-REPLY event eventually causes a GAIN-POSSESSION event. One of the easiest ways to implement this is to use a single queue to implement a first-in first-out disciple on processes waiting after a SERIALIZER-REQUEST or RESOURCE-REPLY event. An acceptable alternative is to give priority to processes waiting after RESOURCE-REPLY events since removing them from the crowd of the resource which is producing the reply might allow other processes (which are waiting for that crowd to empty) to proceed.

III. SERIALIZER CONSTRUCTS

In this section we present the language constructs used in the serializer mechanism. We should note that while a "Lisp-like" syntax is used, we regard the choice of syntax as minor. R. Atkinson is currently developing an "Algol-like" syntax for the serializer construct.

A. Creation

A serializer is constructed by an expression of the form

```
(create_serializer
    (queues: queues_for_the_serializer)
    (priority_queues: priority_queues_for_the_serializer)
    (crowds: crowds_for_the_serializer)
    (entry: entry_of_the_serializer))
```

The queues are used to provide first-in first-out service to process waiting for some condition in order to continue execution. The priority queues are used to provide priority service to processes waiting for some condition so that the service is first-in first-out within priority. In this paper we will use nonnegative integers as priorities. Conceptually a priority queue is a vector of ordinary queues where the index of a queue in the vector is its priority. The crowds are used to record which protected resources are in use.

If a serializer constructed by an expression of the form given above is sent a message M in a SERIALIZER-REQUEST event then M will eventually be sent to entry_of_the_serializer in an ENTRY event which gains possession of the serializer. At most one process can be in possession of a serializer at one time. The queues and crowds for the serializer relate to its internal working and are explained in greater detail below.

B. Waiting

Queues and priority queues are provided to allow a process to wait until some condition is met before proceeding further. To meet this need serializers provide a *guarantee* command which has the following syntax for waiting in queues:

```
(guarantee the_condition
    (wait_in: the_queue)
    (time_out: time_expression
        the_time_out_handler)
    (then: the_guarantee_body))
```

and the following syntax for priority queues:

```
(guarantee the_condition
    (wait_in: the_priority_queue (priority: the_priority))
    (time_out: time_expression
        the_time_out_handler)
    (then: the_guarantee_body))
```

Conceptually, a process executing a *guarantee* command immediately releases possession of the serializer with a GUARANTEE event. It does not regain possession and continue with execution of the_guarantee_body with an ESTABLISHED event unless the following prerequisites hold:

1) the_condition is true;

2) a JOIN-CROWD, EXIT, or GUARANTEE event has just occurred releasing possession of the serializer.

If the process is waiting in a queue then the following additional prerequisite must be satisfied:

3) the process is at the front of the queue.

On the other hand if the process is waiting in a priority queue then the following prerequisite must be satisfied:

3) the process is at the front of the nonempty queue with highest priority (within the priority queue specified in the *wait_in* clause of the *guarantee* command).

Note that all three of these prerequisites must be simultaneously satisfied before execution will continue with the_guarantee_body.

The only other way that a process waiting for a condition can continue execution is that the amount of time it has spent waiting is greater than the amount of time which it specified in the time_expression of the *time_out* clause of the guarantee command. In this case execution of the process continues in the_time_out_handler.

If the guarantee condition for some process waiting at the front of a queue holds or if a process has timed out, then one such process is guaranteed to regain possession of the serializer next before any further ENTRY or LEAVE-CROWD events occur.

The condition in the *guarantee* command may be any Boolean expression without side effects. Conceptually the condition of each process at the front of each queue or priority queue is evaluated whenever possession of the serializer is yielded. In practice we believe that compilers can often produce equivalent code that is much more efficient that this.

We have found one particular form of expression to be quite useful, which is a test for emptiness of queues, priority queues, or crowds. This is written as:

```
(empty: . . . queue_i . . . crowd_j . . .)
```

Each queue, priority queue, or crowd listed must be empty for the expression to be true. The evaluation of an expression of the above form has no side effects. It simply calculates the Boolean value of the expression.

C. Relaying Messages

Within a serializer it is necessary to be able to temporarily release possession of the serializer in order to relay a message to a protected resource and then later regain possession with the reply from the protected resource. It is possible for one serializer to protect more than one resource. A command of the following form accomplishes this by transmitting a_message to a_protected_resource:

```
(relay_to a_protected_resource a_message
   (thru: a_crowd)
   (then_to: continuation_for_reply))
```

To execute a *relay_to* command: place an element in a_crowd to record the presence of another process in a_protected_resource; release possession of the serializer; and then relay a_message to a_protected_resource. After the reply to a_message has been received and possession of the serializer has been regained by a LEAVE-CROWD event, then the element which was placed in a_crowd is removed. The reply received is sent to continuation_for_reply. For brevity we adopt the convention that an expression of the following form:

```
(relay_to a_protected_resource a_message
   (thru: a_crowd))
```

is an abbreviation for

```
(relay_to a_protected_resource a_message
   (thru: a_crowd)
   (then_to: identity_function))
```

where identity_function is (λx. x). Thus the value of an abbreviated *relay_to* command is the value returned as a reply from a_protected_resource as a result of sending it a_message.

During the time between the JOIN-CROWD and the LEAVE-CROWD events of a given process, there is an element in a_crowd for each process using the resource. Thus by inspecting the various crowds of a serializer it is possible to determine which resources currently have processes executing within them.

D. Combined Guarantee and Relay

Expressions of the following form

```
(guarantee condition
   (wait_in: waiting_room)
   (then:
      (relay_to protected_resource message
         (thru: crowd))))
```

occur sufficiently often in serializers that it is worthwhile to have the following abbreviation:

```
(guarantee condition
   (wait_in: waiting_room)
   (relay_to: protected_resource message)
   (thru: crowd))
```

IV. Mutual Exclusion

One of the most common uses of semaphores is to implement mutual exclusion of execution in protected resources. It is relatively easy to implement mutual exclusion using a semaphore. The idea is for each process to perform a *P* operation on the semaphore before using the resource and then to perform a *V* operation when finished using the resource. The program **one_at_a_time** given below can be used to construct systems that insure that a resource does not receive any messages while still processing a previous message. Thus processes are guaranteed to exclude each other from overlapping execution in the protected resource. This simple example is presented to illustrate more concretely the concept of encasing a resource in a serializer.

The implementation below can be used to enforce mutual exclusion in the use of a resource. The character = is used as a prefix operator in front of an identifier which is to be bound to the object which it matches. As in the lambda calculus there are no side effects in performing this binding operation.

```
(one_at_a_time =resource) ≡    ;mutual exclusion of a resource is enforced by
  (let {(mutex = (create_binary_semaphore))}    ;constructing a new binary semaphore called mutex
    (receive=a_message    ;then returning an actor such that whenever it receives a message
      (P mutex)    ;performs a P operation on mutex
      (let {(result = (send a_message to resource))}    ;then sends the message to the resource
        ;such that after the result is received
        (V mutex)    ;a V operation is performed on the semaphore
        result)))    ;and the result is returned
```

In order to guarantee that a resource 𝔭 (assumed to be an actor that receives and replies to messages) is never used by more than one process at a time, construct an actor 𝔭' by invoking (one_at_a_time 𝔭) and only give the actor 𝔭' to potential users. The actor 𝔭' behaves just like 𝔭 except that at most one process can be using 𝔭 at any one time. If the actor 𝔭 is never given to potential users, then the serializer created by invoking **one_at_a_time** will provide the required scheduling discipline.

Semaphores are a very primitive synchronization method which can be used to implement the facilities needed by modular schedulers. Serializers abstract the control structure of schedulers such as the simple one presented above. They can be used to increase the modularity of implementations by making the structure of the implementation more closely match the structure of the task to be accomplished. In this way the synthesis of schedulers from specifications is facilitated because serializers provide facilities for directly implementing common aspects of specifications for schedulers. Furthermore proofs that implementations satisfy their specifications are facilitated because the structure of the serializer guarantees many properties of the implementation that would otherwise have to be painfully extracted from a global analysis of the implementation.

One extremely common specification is that a resource must reply to each request it receives (a guarantee of service that implies that the resource is starvation-free and thus free of deadlock). For serializers this is expressed in terms of events by simply requiring that for every SERIALIZER-REQUEST event there is a corresponding SERIALIZER-REPLY event in the history. Similarly the resource can be required to reply to requests by specifying that for every RESOURCE-REQUEST event there is a corresponding RESOURCE-REPLY event in the history. A **one_at_a_time** is guaranteed to reply to requests provided that the resource it encases is guaranteed to reply to requests.

V. Readers–Writers Implementations

A. Readers-Writers Integrity Specification

A readers–writers serializer is intended to protect the integrity of a resource by scheduling access to the resource in such a way that it is impossible for two processes to overlap in their use of the resource if one of them is a writer. An event in which a message is received by the protected resource is called a RESOURCE-REQUEST, of which there are two special cases:

```
(one_at_a_time =resource) ≡      ;to enforce mutual exclusion for a resource
   (create_serializer      ;create a serializer
      (entry:     ;such that when entry is gained to the serializer
         (receive=a_message      ;with a message
            (send a_message to resource))))    ;send the message to the resource
```

RESOURCE-READ and RESOURCE-WRITE. In response to these requests the protected resource will produce responses which will be called RESOURCE-REPLY events.

The integrity specification for a readers–writers serializer is that "a write activity cannot be concurrent with another read activity or with another write activity." This integrity specification can be expressed in event terms as follows: if SERIALIZER-WRITE$_i$ and SERIALIZER-REQUEST$_j$ are two requests received by the serializer then either

$$\text{RESOURCE-WRITE}_i \longrightarrow \text{RESOURCE-REPLY}_i \longrightarrow \text{RESOURCE-REQUEST}_j \longrightarrow \text{RESOURCE-REPLY}_j$$

or

$$\text{RESOURCE-REQUEST}_j \longrightarrow \text{RESOURCE-REPLY}_j \longrightarrow \text{RESOURCE-WRITE}_i \longrightarrow \text{RESOURCE-REPLY}_i$$

B. Readers-Writers Scheduling Specifications

Variations of the readers–writers problem derive from the desirability of imposing stronger scheduling specifications than simply that the serializer must reply to requests that it receives. Note that readers do not interfere with one another even if they are executing in parallel in the protected resource. Therefore allowing multiple readers into the resource concurrently can increase throughput. Several variations of scheduling specifications that require more concurrency will be presented below. In all of these implementations we will keep track of whether there are readers in the resource or there is a writer in the resource by keeping a separate crowd for the readers and a separate crowd for the writer (which will never have more than one member).

The fundamental integrity constraint for mutual exclusion of the use of a resource is that if two distinct requests SERIALIZER-REQUEST$_i$ and SERIALIZER-REQUEST$_j$ are made to the serializer then either the i-use completely precedes the j-use

$$\begin{tikzcd}
\text{RESOURCE-REQUEST}_i \arrow[d] & \\
\text{RESOURCE-REPLY}_i \arrow[r] & \text{RESOURCE-REQUEST}_j \arrow[d] \\
& \text{RESOURCE-REPLY}_j
\end{tikzcd}$$

or the j-use completely precedes the i-use

$$\begin{tikzcd}
& \text{RESOURCE-REQUEST}_j \arrow[d] \\
\text{RESOURCE-REQUEST}_i \arrow[d] & \text{RESOURCE-REPLY}_j \arrow[l] \\
\text{RESOURCE-REPLY}_i &
\end{tikzcd}$$

which says that one process must enter and leave the protected resource before the other enters.

Actually the serializer version of **one_at_a_time** implements a stronger specification: namely that if

$$\text{SERIALIZER-REQUEST}_i \xrightarrow{-} \text{SERIALIZER-REQUEST}_j$$

then

$$\begin{tikzcd}
\text{RESOURCE-REQUEST}_i \arrow[d] & \\
\text{RESOURCE-REPLY}_i \arrow[r] & \text{RESOURCES-REQUEST}_j
\end{tikzcd}$$

Below we will present several implementations of the readers-writers problems in which **readers** is the crowd of readers in the protected resource and **writer** is the crowd of writers in the protected resource. The following invariants are relevant to understanding each of the implementations to be presented below:

the size of the **writer** *crowd is never greater than one*
the **readers** *crowd and the* **writer** *crowd are never both nonempty at the same time.*

The proofs of the above invariants are relatively short and simple. No complicated chains of reasoning are required. This is an example of how the structure of a serializer enables simpler proofs than less structured arbitration mechanisms such as semaphores.

The proof that the size of the **writer** crowd is never greater than one follows immediately from the observation that the only addition to the **writer** crowd is made after a guarantee is satisfied that the **writer** crowd is empty, and the **writer** crowd is initially empty. By examining the guarantees we see that no additions can be made to the **writer** crowd if the **readers** crowd is nonempty, and that no additions can be made to the **readers** crowd if the **writer** crowd is nonempty. Given that both crowds are initially empty then the **readers** crowd and the **writer** crowd are never both nonempty at the same time. The proofs of these invariants for the other implementations are similar and will not be repeated. Thus the following invariants hold after each event in which a process gains or releases possession of the serializer:

(size writer) $\leqslant$ **1**
(*empty:* **writer**) $\vee$ (*empty:* **readers**)

That is, the above implementation guarantees that writers will exclude others from the resource since if there is an element in the **writer** crowd then all the queues of the serializer are blocked.

C. First-Come First-Served

In the implementation below we provide that a resource which is to be scheduled for reading and writing will receive messages of the form (*read* (*using:* **directions**)) and (*write* (*using:* **directions**)). The **directions** included in these messages can be as complicated as desired up to and including a procedure for carrying out the transaction on the protected resource. It is the responsibility of the protected resource to ensure that carrying out the **directions** in a *read* message does not change the protected resource. Note that this degree of generality in the **directions** can complicate verifying that the resource will reply to each request which it is sent. Nevertheless, in the discussion below we will assume that the resource will always reply to requests of the form (*read* (*using:* **directions**)) and (*write* (*using:* **directions**)).

The implementation given below satisfies the specification that the protected resource is served on a first-come first-served basis. In addition, starvation is not possible and a certain amount of concurrency is guaranteed. The simplicity of the implementation is due to the ability of serializers to have processes waiting in a *single* queue for different conditions.

It is quite easy to see that starvation is impossible for a **first_come_first_served** serializer provided that the resource always replies to messages which it sent. The only way in which starvation could occur would be for some process to wait forever in the **waiting_q**. Thus to prove that starvation is impossible it is sufficient to prove that the front element of the **waiting_q** is always eventually dequeued since the queues of a serializer are strictly first-in first-out. If the element at the front of the **waiting_q** is a reader then it will eventually proceed since the only condition which prevents this is for there to be an element in the **writer** crowd. The **writer** crowd must eventually become empty because the resource is as-

```
((first_come_first_served =the_resource) ≡
;a first come first served serializer of the resource which can be implemented by constructing
  (create_serializer    ;a serializer which has
    (queues: waiting_q)     ;a queue called the waiting_q
    (crowds: readers writer)     ;and two crowds called readers and writer
    (entry:     ;after entry
      (message-cases     ;there are two cases for the message
        ((read (using: =directions))  ⟶  ;receive a request to read the resource using directions
          (guarantee (empty: writer)     ;guarantee that there is no writer in the resource
            (wait_in: waiting_q)
            (relay_to: the_resource (read (using: directions)))
            (thru: readers)))     ;passing thru the readers crowd
        ((write (using: =directions))  ⟶  ;receive a request to write in the resource using directions
          (guarantee (empty: readers writer)
          ;guarantee that there are neither readers nor a writer in the resource
            (wait_in: waiting_q)     ;wait in waiting_q
            (relay_to: the_resource (write (using: directions)))
            (thru: writer))))))     ;passing thru the writer crowd
```

sumed to always respond to write requests. After the **writer** crowd becomes empty then the reader at the front of the **waiting_q** must proceed because serializers give absolute priority in giving next possession of the serializer to processes waiting in queues for some condition to be established. In a similar way it is easy to prove that if the element at the front of the **waiting_q** is a writer then it will eventually proceed.

D. Writers Priority

The following implementation of a serializer for the readers-writers problem gives priority to writers.

```
  ((writers_priority =the-resource) ≡    ;to create a writers priority serializer for a resource
     (create_serializer    ;create a serializer
        (queues: reader_q waiting_q)    ;with two queues called reader_q and waiting_q
        (crowds: writer readers)    ;and two crowds called writer and readers
        (entry:    ;on entry to the serializer
           (message-cases    ;there are two cases for the message
              ((read (using: =directions))  →   ;receive a request to read the resource using directions
#                (guarantee (or (empty: readers) (empty: waiting_q))
                 ;guarantee that there are no readers in the resource or that the waiting_q is empty
                    (wait_in: reader_q)
                    (then:
                       (guarantee (empty: writer)    ;guarantee that the writer crowd is empty
                          (wait_in: waiting_q)    ;wait in the waiting queue
                          (relay_to: the_resource (read (using: directions)))
                          ;then relay the message to the resource
                          (thru: readers)))))    ;passing thru the readers crowd
               ((write (using: =directions))  →   ;receive a request to write in the resource using directions
*                 (guarantee (empty: readers writer)
                  ;guarantee that there are neither readers nor a writer in the resource
                     (wait_in: waiting_q)    ;wait in the waiting_q
                     (relay_to: the_resource (write (using: directions)))
                     (thru: writer)))))))
```

If the **readers** and **writer** crowd are both empty then it is possible for a reader process to be ready to proceed from the *guarantee* command marked with a # at the same time that a writer process is ready to proceed from the *guarantee* command marked with a *. Nevertheless there is no danger of starvation in this case because the writer process at the front of the **waiting_q** and every process in the **reader_q** is guaranteed to proceed before any other process can gain possession of the serializer by the rule of absolute priority for processes waiting for a condition. In view of this example, we propose the following rule for the construction of serializers:

If there is more than one process waiting for a condition that is ready to regain possession of the serializer when possession is released then it must not make any difference in the externally visible behavior of the system which process is chosen to gain possession next.

That is, the choice of which process to allow to gain possession must not cause any change in the ordering relationships on RESOURCE-REQUEST or SERIAL-REPLY events. The above rule allows maximum flexibility to the compiler in evaluating the conditions that must be guaranteed for a waiting process to regain possession of a serializer. However, further research is needed on the question whether the rule is entirely satisfactory for all useful examples.

VI. Behavioral Properties of Serializers

The properties of serializers are stated somewhat informally in this paper since we believe that serializers aid intuitive reasoning about parallelism. A rigorous treatment is possible [13], but is beyond the scope of this paper. Instead in the sections below we systematically state the properties of serials which are used in the proofs in this paper.

Behavioral properties of serializers can be stated in terms of events and relations between events. We shall use the notation

$$E_1 \longrightarrow E_2$$

to indicate that the event E_1 necessarily precedes the event E_2 in the history of the computation under discussion. The precedes relationship is a strict partial order (i.e., it is never the case that there is an event E such that $E \longrightarrow E$). The events of processes that do not interact are not ordered.

In the rest of the paper we will require that protected resources be well-behaved in the sense that for each request sent to a resource exactly one reply will be received.

Another requirement we will make is that every process that comes into possession of a serializer will eventually release possession. The intent is to exclude behaviors where the serializer is locked up forever by a process which is performing an infinite computation while in possession. We believe that this condition will usually be trivial to satisfy in practice.

We have found that the usual code in a serializer to guarantee that some conditions hold and then relay a message to a resource protected by the serializer. This code must be made as efficient as possible in order to maximize the throughput of the serializer. Otherwise the serializer can seriously degrade the efficiency of a system by becoming a bottleneck.

A. Property of Mutual Exclusion

The most fundamental property of a serializer is that at most one process has possession of it at any given instant.

We will use subscripts to indicate distinct invocations of a serializer. The property of mutual exclusion of possession of the serializer can be stated in terms of events as follows where we assume that $i \neq j$:

Either the i-th possession precedes the j-th possession

$$\begin{tikzcd}
\text{GAIN-POSSESSION}_i \arrow[d] & \\
\text{RELEASE-POSSESSION}_i \arrow[r] & \text{GAIN-POSSESSION}_j \arrow[d] \\
 & \text{RELEASE-POSSESSION}_j
\end{tikzcd}$$

or the j-th possession precedes the i-th.

$$\begin{tikzcd}
 & \text{GAIN-POSSESSION}_j \arrow[d] \\
\text{GAIN-POSSESSION}_i \arrow[d] & \text{RELEASE-POSSESSION}_j \arrow[l] \\
\text{RELEASE-POSSESSION}_i &
\end{tikzcd}$$

where a GAIN-POSSESSION$_i$ event is either an ENTRY$_i$, ESTABLISHED$_i$, or LEAVE-CROWD$_i$ event; and RELEASE-POSSESSION$_i$ is the next event after GAIN-POSSESSION$_i$ which is a GUARANTEE$_i$, JOIN-CROWD$_i$, or EXIT$_i$ event.

B. Gaining Possession

We would like to guarantee that any process that sends a request or reply to a serializer must eventually gain possession of the serializer. One way to guarantee this is to prove that an infinite sequence of ESTABLISHED events cannot occur without intervening RELEASE-POSSESSION events. All of the serializers shown in this paper can be easily shown to have this property since there is no iteration or recursion used in the serializers and all of the constructs used by processes in possession of the serializer are known to either terminate or release possession.

If the above restrictions are satisfied then any SERIALIZER-REQUEST or RESOURCE-REPLY event must eventually result in a GAIN-POSSESSION event. More precisely, if there is a SERIALIZER-REQUEST in the history of a computation then it is followed by an ENTRY event. Furthermore if there is a RESOURCE-REPLY event in the history of a computation then it is followed by a LEAVE-CROWD event.

1) First-Come First-Served for Entry: Since serializers are designed to implement scheduling of access to protected resources it must be possible for them to observe the order of arrival of requests to the serializers in order to carry out certain scheduling tasks. Thus we provide that requests for entry into the serializer will be served in the order in which they arrive at the serializer. In terms of events this can be formalized by supposing that SERIALIZER-REQUEST$_i$ and SERIALIZER-REQUEST$_j$ are two events such that the first arrives before the second:

$$\text{SERIALIZER-REQUEST}_i \longrightarrow \text{SERIALIZER-REQUEST}_j$$

Suppose that the process in which SERIALIZER-REQUEST$_i$ occurs continues execution with ENTRY$_i$ and the process in which SERIALIZER-REQUEST$_j$ occurs continues execution with ENTRY$_j$. We require that these two subsequent events be ordered as follows:

$$\text{ENTRY}_i \longrightarrow \text{ENTRY}_j$$

2) First-Come First-Served for Reentry: Similarly it must be possible for a serializer to observe the order of arrival of replies to requests sent to protected resources. Thus if RESOURCE-REPLY$_i$ precedes RESOURCE-REPLY$_j$ so that

$$\text{RESOURCE-REPLY}_i \longrightarrow \text{RESOURCE-REPLY}_j$$

then we require that

$$\text{LEAVE-CROWD}_i \longrightarrow \text{LEAVE-CROWD}_j$$

C. Properties of Guaranteed Conditions

1) The Guaranteed Condition is True if Execution Continues: Let C be the condition guaranteed in an event of the form

$$\text{GUARANTEE}^{C}\text{-WAIT-IN}^{q}$$

which is caused by executing an expression of the form

```
(guarantee C
   (wait_in: q)
   (then: . . .))
```

If a process continues execution after an event of the above form, then the next event of the process is of the form ESTABLISHED^C and C is true at the instant of this event.

2) Queues are First-In First-Out: Suppose that q is an explicit queue of the serializer S and that there are events such that

$$\text{GUARANTEE}^{C1}\text{-WAIT-IN}^{q}{}_i \longrightarrow \text{GUARANTEE}^{C2}\text{-WAIT-IN}^{q}{}_j \longrightarrow \text{ESTABLISHED}^{C2}{}_j$$

Then there is an event ESTABLISHED$^{C1}{}_i$ such that

$$\text{ESTABLISHED}^{C1}{}_i \longrightarrow \text{ESTABLISHED}^{C2}{}_j$$

which says that ESTABLISHED$^{C1}{}_i$ occurred before ESTABLISHED$^{C2}{}_j$ since both guarantee events made use of the same queue.

3) Priority for Processes Waiting for a Condition: A process will be said to be "ready to go" at the instant of a RELEASE-POSSESSION event if it is waiting because of a previous GUARANTEE^C-WAIT-IN^q event, but the corresponding ESTABLISHED event has not yet occurred and the following properties hold:

1) the condition C is true.

2) the process is at the front of the queue. Therefore all previous events that waited for some condition on q have already continued with their condition ESTABLISHED.

This means that we have made the policy decision that for the queues used in this paper a process at the head of a queue will block the processes behind it in the queue even though the condition for one of these processes to proceed is true. We have made this policy decision in order to make the implementation of the serializers in this paper more efficient and to avoid problems of fairness and starvation.

The above properties give the processes waiting in queues and priority queues of a serializer priority over processes waiting to gain possession.

VII. Guaranteed Concurrency

A. Requiring Concurrency in Implementations

In the readers-writers problems the resource scheduler must maintain the following constraint: a write activity on a protected resource is never concurrent with any other activity on the resource. However, a simple one-at-a-time approach can easily guarantee this property. The more complex versions of the problem attempt to provide readers with concurrent access to the resource without starving the writers. When we say that some amount of concurrency is guaranteed, we mean that the specifications for the serializer require that certain readers be given the opportunity to access the resource at the same time.

Note that a serializer cannot guarantee that the requests to a protected resource are actually processed in parallel, since either the structure of the resource or some externally defined scheduling policy may prevent actual parallelism. We say that readers R_i and R_j are **concurrent readers** if

$\text{JOIN-CROWD}_i \longrightarrow \text{LEAVE-CROWD}_j$

and

$\text{JOIN-CROWD}_j \longrightarrow \text{LEAVE-CROWD}_i$

The specifications for the first-come first-serve serializer include a requirement for concurrency. We can informally express this requirement by saying that whenever one reader's entry into the serializer (an ENTRY event) immediately precedes another reader's entry, and the second reader enters the serializer before the first reader enters the resource (a JOIN-CROWD event), then these two readers must concurrently be in the resource. We can also give a more formal specification in terms of events:

If R_i and R_j are readers such that

$\text{ENTRY}_i \longrightarrow \text{ENTRY}_j \longrightarrow \text{JOIN-CROWD}_i$

and there is no requestor X_k (a reader or writer) such that

$\text{ENTRY}_i \longrightarrow \text{ENTRY}_k \longrightarrow \text{ENTRY}_j$

then R_i and R_j must be concurrent readers, i.e.,

$\text{JOIN-CROWD}_i \longrightarrow \text{LEAVE-CROWD}_j$

and

$\text{JOIN-CROWD}_j \longrightarrow \text{LEAVE-CROWD}_i$

Note that the above requirement would be the same if we required that the requestor X_k be a writer, although the proof would be slightly more difficult.

B. Proof of Guaranteed Concurrency

A proof that the **first_come_first_served** serializer shown above satisfies the given concurrency requirement proceeds by assuming the existence of two readers with the given relationship, then showing that they must be concurrent readers. Since we have

$\text{ENTRY}_i \longrightarrow \text{ENTRY}_j \longrightarrow \text{JOIN-CROWD}_i$

we know that the reader R_i must be in the **waiting_q** when the reader R_j gains possession of the serializer. R_j must be enqueued directly behind R_i, since by our assumptions there are no intervening entries to put other requestors in the **waiting_q**. Therefore when reader R_i does get into the resource through a JOIN-CROWD event (thereby releasing possession of the serializer), then the requestor at the front of the **waiting_q** must be R_j and the condition of (*empty:* **writers**) must be true. We then appeal to the priority which serializers give to processes waiting for a condition over other processes which are waiting to gain possession.

$\text{JOIN-CROWD}_j \longrightarrow \text{LEAVE-CROWD}_i$

Since we know by construction of the serializer that $\text{JOIN-CROWD}_i \longrightarrow \text{JOIN-CROWD}_j$, and $\text{JOIN-CROWD}_j \longrightarrow \text{LEAVE-CROWD}_j$, we conclude that $\text{JOIN-CROWD}_i \longrightarrow \text{LEAVE-CROWD}_j$, which completes the proof that R_i and R_j are concurrent readers.

VIII. Absence of Starvation

The following serializer forces readers into the resource concurrently. However, we need to guard against starvation. Our approach is to allow all waiting readers to enter the resource, then to designate the writer which has been waiting as the new privileged writer, and keep further readers from entering the resource until the privileged writer has relayed its message to the resource. After the privileged writer has been served, then all readers which have been waiting for that writer to finish are allowed to enter the resource, and a new privileged writer is chosen. A reader may not deliver a message to the resource while there is a privileged writer, or there is a writer in the resource. A writer may not enter the resource unless it is a privileged writer, and there are neither readers nor a writer in the resource.

```
((readers_priority = the_resource) ≡
  ;a serializer which enforces concurrency among readers of the resource is implemented by constructing
  (create_serializer    ;a serializer which has
    (queues: waiting_q writers_q)    ;two queues called waiting_q and writers_q
    (crowds: readers writer)    ;two crowds called readers and writer
    (entry:    ;after entry
      (message-cases    ;there are two cases for the message
((read (using: =directions))  ⟶  ;receive a request to read the resource using directions
   (guarantee (empty: writer)    ;guarantee that there are no writers in the resource
      (wait_in: waiting_q)    ;wait in waiting_q
      (relay_to: the_resource (read (using: directions)))
      (thru: readers)))    ;passing thru the readers crowd
((write (using: =directions))  ⟶  ;receive a request to write in the resource using directions
   (guarantee (empty: waiting_q writer)
   ;guarantee that waiting_q, and writer crowd are all empty
      (wait_in: writers_q)    ;wait in writers_q
      (then:
         (guarantee (empty: readers writer)
         ;guarantee that there are neither readers nor a writer in the resource
            (wait_in: waiting_q)    ;wait in waiting_q
            (relay_to: the_resource (write (using: directions)))
            (thru: writer))))))))    ;passing thru the writer crowd
```

The above implementation is a little more complicated than the previous one. However, it is not difficult to show that writers exclude others using the technique used for the previous implementation since the following invariants are maintained:

$$(\textbf{size writer}) \leq 1$$
$$(\textit{empty:}\ \textbf{writer}) \vee (\textit{empty:}\ \textbf{readers})$$

In order to show that neither readers nor writers can possibly starve, consider the following "traffic diagram" for the queues and crowds of the serializer:

```
Entering readers              Entering writers
        |                             |
        |                             |
        |                             V
        |                         writers_q
        |                             |
        |                             |
        +---------+    +--------------+
                  |    |
                  |    |  (empty: waiting_q writer)
                  |    |
                  V    V
                 waiting_q
                  |    |
 (empty: writer)  |    |  (empty: readers writer)
                  |    |
        +---------+    +--------------+
        |                             |
        |                             |
        |                             |
        V                             V
     readers                       writer
```

Traffic Diagram for Queues and Crowds of readers_priority

The idea of the proof is to first show that the **waiting_q** must eventually empty, then to show that any writer in the **writers_q** must eventually migrate to the **waiting_q**. These two conditions ensure that every read or write request to the serializer is eventually satisfied.

A. Proof that the waiting_q *Must Empty*

If there is a writer in the **waiting_q**, then there is only one such writer, and it must be at the front of the queue. A writer can only enter the **waiting_q** after it has been dequeued from the **writers_q**, and the guarantee of every writer exiting the **writers_q** is that the **waiting_q** is empty. Thus, not only is it true that there may be only one writer in the **waiting_q**, the writer must also be at the front of the queue if it is there.

Processes may only enter the **readers** crowd or the **writer** crowd by first exiting from the **waiting_q**. We have assumed that **the_resource** has the property that every message sent to the resource will eventually produce a single reply. Therefore, if a writer is at the front of the **waiting_q**, it is guaranteed that no messages will be sent to **the_resource** from the serializer until both the **readers** crowd and the **writer** crowd are empty.

By a similar argument, if a reader is at the front of the **waiting_q**, then there are only readers in the queue. Furthermore the **writer** crowd must eventually empty which implies that the reader at the front of the **waiting_q** must eventually leave. For each reader which leaves the **waiting_q** with an ESTABLISHED event there must be a subsequent JOIN-CROWD event. For every such JOIN-CROWD event, the serializer is released, and if the **waiting_q** has a reader at its front, that reader must be dequeued, since the **waiting_q** is the only explicit queue with its guaranteed condition true (the **writer** crowd remains empty).

Thus we have shown that if there is a writer in the **waiting_q** then it is at the front and that it must eventually leave. Once a writer exits the **waiting_q** there may be no additional writers added to that queue until it is empty. Furthermore, if there is a reader at the front of the **waiting_q** then there are only readers in the queue and they must all eventually leave. Therefore, the **waiting_q** must empty.

B. Proof That No Process in the writers_q *Can Starve*

The idea behind this proof is simple. Any writer in the **writers_q** is waiting for both the **writer** crowd and the **waiting_q** to empty. By our assumption about the resource there must be a RESOURCE-REPLY message for any writer, so the **writer** crowd must empty. We have just proved above that the **waiting_q** must empty. Therefore the process at the front of the **writers_q** must eventually be dequeued, which is sufficient to show that no process in the **writers_q** can starve.

C. Requiring Concurrency for Reader's Priority

The readers-priority serializer is intended to give more throughput to readers at the expense of writers, while still guaranteeing that each write request will receive a reply. Our informal requirement is whenever one reader's entry into the serializer follows another reader's entry (regardless of intervening serializer entries), and the second reader enters the serializer before the first reader enters the resource, then the two readers are concurrent within the resource. In terms of events, this requirement can be expressed as:

If R_i and R_j are readers such that

$\text{ENTRY}_i \longrightarrow \text{ENTRY}_j \longrightarrow \text{JOIN-CROWD}_i$

then R_i and R_j must be concurrent readers, i.e.

$\text{JOIN-CROWD}_i \longrightarrow \text{LEAVE-CROWD}_j$

and

$\text{JOIN-CROWD}_j \longrightarrow \text{LEAVE-CROWD}_i$

Note that the above requirement is stronger than the concurrency requirement given for **first_come_first_served.**

All readers that have entered the serializer and are not yet in the resource are in the **waiting_q**, and the only writer that can be in the **waiting_q** must be at the front of the queue. We have previously shown that once one reader can leave the **waiting_q** then all must leave the **waiting_q** and enter the resource. We note that by the axiom of giving explicit queues priority over implicit queues all readers in the **waiting_q** must enter the resource before any reply from the resource will enter the serializer (through a LEAVE-CROWD event). This completes the proof.

IX. Using Priority Queues

Hoare has used priority queues in monitors to implement a disk head scheduler which optimizes the head motion of a physical disk. Below we present an implementation of a virtual disk which makes the head motion more efficient for the physical disk which is its protected resource. The operations on the virtual disk are exactly the same as the ones defined on the physical disk. The implementation given below uses an algorithm similar to one used by Hoare. However, it will be possible to prove that our implementation using serializers always performs all disk operations which are requested. It is not possible to prove this property for the implementation using monitors because a process might request the disk and then never release it.

```
((virtual_disk =physical_disk) ≡     ;a virtual disk behaves like a physical disk with optimized head motion
  (create_serializer     ;create a serializer with
    (priority_queues: up_queue down_queue)     ;two priority queues
    (crowds: disk_users)     ;one crowd
    (entry:     ;then after gaining possession
      (let
        {(current_direction initially "up")}     ;the current direction of motion is initially up
        ;current_direction is a variable which is local to the serializer
        (message-cases
          ((disk_request (operation: =op) (track: =track_number)) ⟶
            ;receive a disk request with specified operation and track number
            (rules current_direction     ;the rules for the current direction are
              ("down"  ⟶  ;if the current direction is down then
                (guarantee     ;guarantee that
                    (and (empty: disk_users)     ;no one is using the physical disk
                        (or (empty: down_queue)     ;and that either the down queue is empty
                          (current_direction = "up")))     ;or an upsweep is in progress
                  (wait_in: up_queue (priority: track_number))     ;wait in up_queue
                  (then: (current_direction ⟵ "up")     ;then change the current direction to be up
                      (relay_to physical_disk (disk_request (operation: op) (track: track_number))
                      ;relay to the physical disk the message received
                          (thru: disk_users)))))     ;passing thru the disk_users crowd
              ("up"  ⟶  ;if the current direction is up then
                (guarantee     ;guarantee that
                    (and (empty: disk_users)     ;no one is using the physical disk
                        (or (empty: up_queue) (current_direction = "down")))
                    ;and that either the up queue is empty or a down sweep is in progress
                  (wait_in: down_queue (priority: ((number_of_tracks physical_disk) - track_number)))
                  ;wait in down_queue
                  (then: (current-direction ⟵ "down")     ;then change the current direction to be down
                      (relay_to physical_disk (disk_request (operation: op) (track: track_number))
                      ;and relay the message to the physical disk
                          (thru: disk_users)))))))))))
```

In proving that the virtual disk satisfies its specifications we need to show that the requests are relayed though unchanged, that the serializer only allows one process at a time access to the physical disk, and that all requests given to the virtual disk are eventually serviced.

The relaying of unchanged requests can be shown by a simple examination of the code for the virtual disk. Only one process may be using the physical disk at a time since both guarantees require that the crowd for the physical disk be empty before any requests are allowed to proceed.

Proving that all requests are eventually serviced is done by showing that all requests in the queue for the current direction (the **up_queue** when **current_direction = "up,"** the **down_queue** when **current_direction = "down"**) are eventually serviced, and that the opposite queue becomes the current queue. Note that the **current_direction** is reversed if and only if the current queue is empty and the opposite queue is not empty and the physical disk is not busy. A new request is never added to the current queue. We assume that the physical disk always completes a request. Whenever the physical disk completes a request and the current queue is not empty the guarantee for the current queue must be true. Therefore all requests in the current queue must eventually be serviced, and the opposite queue must become the current queue (or be empty).

Requests removed from a priority queue must be in order of increasing priority number (nondecreasing track number for **up_queue**, nonincreasing track number order for **down_queue**). This is sufficient to show that requests within the current queue are always serviced with minimal difference in track numbers. However, it does not guarantee minimal head motion for all requests.

A somewhat more optimal version of **virtual_disk** which more closely follows Hoare's original disk scheduler can be achieved by allowing insertion of new requests into the current queue as long as the priority is greater than the priority of the current request. Additional code in the serializer is needed in that a current track number must be maintained and used to determine which queue should be used. Additional clauses in the proof are also needed to show that all requests are serviced (by showing that all requests in the current queue have priorities greater than the priority of the requests being serviced). However, we consider it encouraging that both the program and the proof are only incrementally more difficult, and that neither is different in kind.

X. Comparison with Monitors

The serializer mechanism is a more structured version of the "secretary" concept which was conceived in general terms by Dijkstra and later developed into a specific programming language construct called "monitors" by Brinch Hansen and Hoare. The purpose of serializers is to schedule access to shared resources in order to protect their integrity. A serializer is an actor that will allow only one process to have possession at a time whereas a monitor is a Simula-67 class that will allow only one process to be executing inside it at a time. We claim that serializers support modular programming better than monitors because serializers can be sensibly nested inside one another, whereas usually it is unprofitable to nest one monitor inside another because the outer monitor will be tied up while the inner one is in use. A further advantage for serializers is that use of a protected resource appears identical to the use of the unprotected resource in a program that uses the resource. A general principle of efficient operation that is applicable to both serializers and monitors is to try to keep the serializer (monitor) unlocked as much of the time as possible to keep it from being a bottleneck in the operation of a larger system.

We believe that serializers are easier to write and verify in a modular fashion than monitors. Serializers encapsulate useful properties of the use of shared resources that are more difficult to verify with schedulers written using monitors. For example it is easy to verify that processes which use the virtual disk resource implemented in this paper do not indefinitely tie up the physical disk, whereas this property cannot be proved using the diskhead monitor presented in Hoare's paper.

Using serializers the condition necessary for a process to continue execution is explicitly stated in a *guarantee* command, whereas the "conditions" used in the WAIT command in a monitor do not explicitly state what condition is necessary in order to proceed. In our experience this feature of serializers lessens the number of explicit queues and tends to simplify program proofs.

Another difference is the use of crowds rather than counters to keep track of processes that have been allowed to access the encapsulated resource. While there is an additional cost associated with such accounting, we believe that the benefits will make the use of crowds a decided advantage. It is possible to examine the crowds to determine which processes are currently accessing which resources.

XI. Future Work

A. Efficiently Compiling Serializers

On the basis of the examples that we have analyzed we believe that serializers can be efficiently compiled to produce code for synchronizers that approaches the efficiency of other proposed synchronization primitives such as semaphores, monitors, conditional critical regions, and path expressions. Our analysis indicates that schedulers constructed using these other synchronization primitives are in practice going to have to test essentially the same conditions that are explicit in the *guarantee* commands of serializers. The efficiency of serializers needs to be tested by constructing a compiler.

B. Prohibiting Starvation

The simple specification that "starvation is prohibited" is common to almost all synchronization specifications. Yet serializers at present make no such guarantee. They do make it easier to prove that an implementation using a serializer is free of the danger of starvation by facilitating the proof that the serializer always replies to requests which it receives. We feel that research should continue to search for mechanisms that provide effective guarantees of such properties, yet also provide sufficient generality to cope with a wide range of problems.

C. Preempting Processes in Protected Resources

Serializers have potential for use in robust systems in that more information is available for error recovery. The additional information is useful for implementing debugging features, deadlock detection, and gracefully backing processes out of protected resources.

One proposal [8a], [19a] is to keep a log of the side effects of a process executing in a protected resource. A runaway process in a protected resource could be preempted without destroying the integrity of the resource by undoing all side effects recorded in the log. Keeping such a log would be a significant source of overhead which would only be worthwhile if it significantly increased the reliability or efficiency of systems. For example it would enable us to generalize the *relay_to* command by the addition of a time out clause as follows:

```
(relay_to a_protected_resource a_message
  (thru: a_crowd)
  (time_out: time_expression
     the_time_out_handler))
```

where the_time_out_handler is invoked if time_expression expires before a_protected_resource has replied to a_message.

D. Generality of Serializers

The generality of serializers (monitors, path expressions, etc.) still needs to be tested on a large number of nontrivial applications. Showing that serializers are a secure, modular, efficient mechanism for implementing all the versions of the readers-writers problem and the disk head scheduling problem is no guarantee that they can do as well for other scheduling problems that arise in large real time systems.

XII. Conclusions

In this paper we have introduced a modular arbitration primitive called a *serializer* which is a more structured version of the monitor construct previously developed by Brinch Hansen and Hoare. Serializers aid in the synthesis of modular synchronizers because their structure corresponds in a natural way to typical specifications for useful synchronizers. The structure imposed by using serializers provides important guarantees that aid in proving that the implementation meets its specifications.

The specifications for a serializer include integrity specifications relating the order of access to the type of access, and scheduling specifications to ensure that differing types of access occur in the proper order. Part of this ordering specification included a specification requiring that certain requestors must be given the opportunity to use the resource concurrently. In the readers-writers serializer, we gave the integrity specifications that readers and writers were mutually exclusive in accessing the resource, and that at most one writer could access the resource at a time. Our different solutions to the readers-writers problem resulted from different scheduling specifications.

We have attempted to explicitly introduce facilities into serializers that directly correspond to synchronization specifications. The constraint that the resource is not being used by either readers or writers when a writer enters the resource is explicit in the code of our implementations, as is the requirement that no writers are using the resource when a reader enters. Serializers provide that the condition for a waiting process to proceed is explicitly stated. In this way integrity specifications can be directly expressed in the language. Scheduling specifications are more complicated. We have been able to use a specification language based on partial orders among events to good effect to express scheduling specifications. Furthermore, the structure of the serializers has enabled us to give simple intuitive proofs that various scheduling specifications are satisfied by implementations that use serializers.

Acknowledgment

This paper builds on the thesis work of I. Greif who gives behavioral specifications for all of the versions of the readers-writers problems given in this paper. She presents scheduling specifications for the readers-writers problem that are very similar to the ones in this paper. A. Yonezawa [23] has developed improved notations for expressing specifications like the ones in this paper. D. Reed is also investigating the use of time outs for concurrent processes using shared resources in a distributed system.

Conservations with O.-J. Dahl, E. W. Dijkstra, T. Hoare, J. H. Morris, and J. Reynolds have helped us to crystallize our ideas on these issues. T. Hoare, M. Laventhal, D. Brock, and W. Clinger made suggestions that have materially improved the content of this paper. We would particularly like to thank one dedicated referee who carefully read through the paper and suggested numerous improvements in presentation.

The concept of a serializer which we have developed owes a tremendous intellectual debt to the monitors developed by Brinch Hansen and Hoare. We have attempted to build on their work to develop better structured arbitration primitive for actor message passing systems. The development of serializers would not have been possible if monitors had not been previously developed. Brinch Hansen has developed the programming language Concurrent Pascal which includes monitors. He has used the language in the design and implementation of an extremely modular single-user operating system called Solo. D. Bustard and T. Hoare at the Queen's University of Belfast have participated in the development of the programming language Parallel Programming Pascal which includes a mechanism called an *envelope*, which is related to our *serializer* mechanism.

In December 1975 a very preliminary version of this paper was extensively revised to reflect improvements in serializers that were achieved in the PLASMA implementation meetings that fall in which R. Atkinson, M. Freiling, C. Hewitt, K. Kahn, H. Lieberman, M. McLennan, R. Pankiewicz, B. Smith, and A. Yonezawa were active participants. These meetings resulted in substantial improvements in our understanding of modular arbitration mechanisms which were presented by the authors

at the January 1976 Principles of Programming Languages Conference in a paper entitled "Synchronization in Actor Systems."

We originally were inspired to investigate the message-passing paradigm for programming languages by a lecture which A. Kay delivered at the MIT Artificial Intelligence Laboratory in November 1972 on a preliminary version of the SMALLTALK-72 language [22] which has subsequently evolved into SMALLTALK-76 [21] and [20]. The SMALLTALK work in turn builds on Simula-67 [2]. This paper and companion papers [1], [13] attempt to extend this programming language paradigm into the realm of concurrent programming.

Recently [13a] the ideas in this paper have been carried an important step forward with the development of "primitive serializers" for implementing guardians in distributed systems. Communicating Sequential Processes [19] have been developed which also attempt to generalize the communication paradigm to concurrent programming. A serializer imposes a total ordering in time for all messages that arrive. This enables us to formally treat issues of starvation and fairness in a relatively straightforward way in this paper. As far as we know, methods for proving these properties have not yet been developed for Communicating Sequential Processes.

References

[1] H. J. Baker, Jr. and C. Hewitt, "Incremental garbage collection of processes," MIT Artificial Intelligence Memo 454, Dec. 1977.

[2] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*. New York: Auerbach, 1973.

[3] P. Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199–207, June 1975.

[4] —, "The solo operating system," *Software-Practice and Experience*, pp. 141–205, Apr.–June 1976.

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

[6] D. W. Bustard, "Parallel Programming Pascal (PPP)," version 1, Dept. Comput. Sci., Queen's Univ. Belfast, Nov. 1975.

[7] E. W. Dijkstra, "Co-operating sequential processes," in *Programming Languages*, F. Genuys, Ed. New York: Academic, 1968.

[8] —, "Hierarchical ordering of sequential processes," *Acta Informatica*, 1971.

[8a] K. P. Eswaren, 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. 11, pp. 624–633, Nov. 1976.

[9] N. Goodman, "Coordination of parallel processes in the actor model of computation," MIT Lab. Comput. Sci. TR 173, June 1976.

[10] I. Greif, "Semantics of communicating parallel processes," MAC Tech. Rep. TR-154, Sept. 1975.

[11] I. Greif and C. Hewitt, "Actor semantics of PLANNER-73," in *Proc. Ass. Comput. Mach. SIGPLAN-SIGACT Conf.*, Palo Alto, CA, Jan. 1975.

[12] C. Hewitt and R. Atkinson, "Synchronization in actor systems," in *Proc. Conf. Principles of Programming Languages*, Los Angeles, CA, Jan. 1977.

[13] C. Hewitt and G. Attardi, "An axiomatic denotation specification of a concurrent programming language," MIT Working Paper, May 1978.

[13a] C. Hewitt, G. Attardi, and H. Lieberman, "Specifying and proving properties of guardians for distributed systems," MIT AI Working Paper, Oct. 1978.

[14] C. Hewitt and B. Smith, "Towards a programming apprentice," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 26–45, Mar. 1975.

[15] C. Hewitt and H. Baker, "Laws for communicating parallel processes," MIT Artificial Intelligence working Paper 134, Dec. 1976; Invited paper presented at IFIP 1977.

[16] —, "Actors and continuous functionals," presented at IFIP Working Conf. Formal Description of Programming Concepts, Aug. 1–5, 1977, St. Andrews, New Brunswick, Canada; MIT Artificial Intelligence Memo 436A, July 1977.

[17] C. A. R. Hoare, "Towards a theory of parallel programming," in *Operating Systems Techniques*. New York: Academic, 1972, pp. 61–71.

[18] —, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, Oct. 1975.

[19] —, "Language hierarchies and interfaces," in *Lecture Notes in Computer Science*, no. 46. New York: Springer, 1976, pp. 242–265.

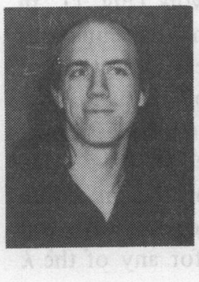
[19a] B. W. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," *Commun. Ass. Comput. Mach.*, to be published.

[20] D. H. H. Ingalls, "The Smalltalk-76 programming system design and implementation," in *Conf. Rec. 5th Annu. ACM Symp. Principles of Programming Languages*, Tuscon, AZ, Jan. 23–25, 1978, pp. 9–16.

[21] A. Kay, "Microelectronics and the personal computer," *Scientific American*, Sept. 1977.

[22] J. F. Shoch, "An overview of the programming language Smalltalk-72," presented at the Convention Informatique 1977, Paris, France.

[23] A. Yonezawa, "Specification and verification techniques for parallel programs based on message passing semantics," MIT Ph.D. dissertation, Dec. 1977; MIT Lab. Comp. Sci. Tech. Rep. 191.



Carl E. Hewitt was born in Clinton, IA, on December 12, 1944. He received the B.S. degree on a McDermott Scholarship, and the Ph.D. degree on a Fellowship, both in mathematics, from the Massachusetts Institute of Technology, Cambridge, in 1967 and 1971, respectively.

His graduate work was concerned with artificial intelligence and theories of computation. He is presently an Assistant Professor of Computer Science, Massachusetts Institute of Technology.

Russell R. Atkinson was born in New York City, NY, on August 17, 1950. He received the S.M. degree from the Massachusetts Institute of Technology, Cambridge, MA, in 1976.

His graduate work has been concerned with programming languages. He is currently a Ph.D. candidate at the Massachusetts Institute of Technology.