3-1-1981

Specifying and Proving Properties of Sentinels

Krithivasan Ramamritham
University of Massachusetts - Amherst

Robert M. Keller
Harvey Mudd College

Recommended Citation

Ramamritham, K., and R.M. Keller. "Specifying and proving properties of sentinels." Proceedings of the 5th International Conference on Software Engineering (March 1981): 374-382.

SPECIFYING AND PROVING PROPERTIES OF SENTINEL PROCESSES(*)

Krithivasan Ramamritham and Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

This paper presents a technique for specifying and verifying properties of "sentinels" -- a high-level language construct for synchronizing access to shared resources. Statements in the specification language possess formal temporal semantics. As a prelude to proving the correctness of sentinels, the semantics of constructs used in sentinels is given. The proof technique involves showing that the temporal behavior of a sentinel conforms to that defined by the specification. The methodology is illustrated by applying it to a typical synchronization problem.

INTRODUCTION

We are concerned with the problem of synchronizing access to shared resources by concurrently executing processes. In particular, we are interested in the specification of synchronization among processes as well as a methodology for verifying that a given synchronizer possesses specified properties. The specification and verification techniques proposed are founded on temporal logic[17], the chief advantage of which is that it facilitates a unified approach to specification and verification of both invariant and time-dependent properties of software systems. When one refers to ordering of operations, scheduling discipline, etc., the underlying concept is temporal ordering. Thus it is appropriate to adopt a system of reasoning based on temporal logic for expressing the semantics of, and for validating, the synchronization of concurrent processes.

Our specification language is designed to express various aspects of synchronization control, such as constraints governing access to shared resources, priority of various types of access, mutual exclusion of access, invariance of the resource state, absence of starvation, and other relevant properties. Statements in the language use the primitive temporal constructs "henceforth", "eventually" and "until", and other constructs that can be expressed in terms of the primitives. Each statement has appropriate formal temporal semantics. Details of the specification language can be found in[18].

Here we confine our attention to showing the correctness of a synchronizer that has been specified in this language. The proof technique is based on the following key observation: The temporal behavior of a synchronizer of concurrent processes induces a temporal behavior in the synchronized processes. Verifying the synchronizer then reduces to showing that the behavior induced by the synchronizer conforms to the specified behavior. Thus the proof technique has two phases:

1. The temporal behavior induced by the synchronizer is determined from the synchronization code, using the temporal semantics of the constructs.

2. The induced behavior is shown to imply the specified behavior, by applying theorems of temporal logic.

The first phase is similar to the inductive assertion method[4] for verifying sequential programs.

In this paper we demonstrate the approach by applying it to sentinels[13]. A sentinel is a sequential process which coordinates other processes by using queueing primitives to provide a basic form of synchronization. These primitives allow sentinels to exchange data with, and control execution of, the processes being coordinated. Besides providing a means to program interacting concurrent processes in a structured manner, sentinels can be used to achieve highly-tailored disciplines for coordination among processes. A typical synchronization problem is used throughout the paper to illustrate use of the specification language and the proof method, and to demonstrate that a unified approach can result from using temporal logic as a semantic basis.

(*)This material is based upon work supported by the National Science Foundation under grant MCS 77-09369.

A LANGUAGE FOR SPECIFYING SYNCHRONIZATION

First we explain our concept of "synchronization". A synchronizer (of which sentinels are examples), is a sequential process that guarantees disciplined access to shared resources. Access to a shared resource is through specific operations, the execution of which is controlled by the synchronizer. Constraints essential for maintaining the integrity of the resource are enforced by the synchronizer.

Concurrent processes can access the shared resource by requesting execution of any of the specified operations. A request for an operation on a shared resource is serviced by the synchronizer after ensuring that none of the constraints is violated. A serviced request becomes active when it is executed either by the synchronizer or, on its behalf, by another process. This model assumes that

1. There may be a finite delay between servicing a request and its subsequent activation.
2. An active process cannot be aborted or interrupted.
3. An operation remains active for a finite but indefinite period of time, after which it is said to have terminated.

These assumptions will be formalized after the introduction of the language primitives.

Specification Language Primitives

We refer to each distinct type of operation on a shared resource as an operation class. All operations of a particular type are said to be instances of that operation class. In the definitions below, "a" stands for a specific instance of a particular operation class.

Every pending instance has associated with it four primitive conditions with the following semantics.

Req(a)	This condition becomes true when a concurrent process requests operation "a", and remains true until the requested operation is serviced.
Start(a)	This condition becomes true when the synchronizer services request "a" and remains true until "a" starts execution.
Exec(a)	This condition is true when execution of operation "a" is in progress.
Term(a)	This condition becomes true when operation "a" terminates.

We will now introduce the temporal operators along with their semantics. These are strongly influenced by Lamport[14], Owicki[16] and Pnueli[17].

$\Box$C	To be read "always C". This means condition C will remain true from now on, i.e., C is true now and throughout the future.
$\Diamond$C	To be read "eventually C". This means condition C is true now or will eventually become true.
A UNTIL B	To be read as "A remains true until B becomes true". This means if B eventually becomes true, then A remains true from now until B becomes true; otherwise $\Box$A.

In our language, specifications are statements in first-order predicate calculus augmented with the temporal operators above. Statements in the language may involve the predicate logic operators: V(or), &(and) ~(not) and =>(implication). Certain temporal operators are derived from these primitives, and are introduced to enhance the readability of the specification language. They are,

P ONLYIF Q	(P => Q) i.e., P is true only if Q is true.
P ONLYAFTER Q	(~P UNTIL Q) i.e., P can become true only after Q does.
P TRIGGERS Q	[P => ($\Diamond$Q & P UNTIL Q)] & [~Q => (~Q UNTIL P)] If P is true, Q will become true; P remains true until Q becomes true. If Q is false, it remains false until P becomes true.

P and Q are arbitrary conditions. The following axioms formalize the synchronization model and are asserted for each operation "a".

```
Start(a) => Req(a)
Start(a) TRIGGERS Exec(a)
Start(a) TRIGGERS ~Start(a)
Start(a) TRIGGERS ~Req(a)
Start(a) => [∀b≠a ~Start(b) UNTIL ~Start(a)]
Exec(a)  => [Exec(a) UNTIL Term(a)]
Term(a)  => Exec(a)
Term(a)  TRIGGERS [~Term(a) & ~Exec(a)]
```

For notational convenience, we introduce the following predicates.

Req$A	∃a∈A Req(a), i.e., there exists a request of class A.
Exec$A	∃a∈A Exec(a), i.e., an operation of class A is active.

The temporal operators defined earlier serve as building blocks for our specification language. The semantics of the various specification statements are given in terms of these temporal operators. The reader can refer to [18] for a detailed description of the specification language.

To highlight various features of the specification language, sentinels, and the proof technique, we introduce the following synchronization problem: Two classes of operations, namely "Low" and "High", access a shared resource. High operations are required to exclude each other, while a low operation may execute concurrently with other operations. However, maximum possible concurrent low operations is limited by "maxavail". "Inuse" gives the number of low operations currently in execution. In addition, we want to expedite servicing high requests and hence they are given priority over low requests. Below is the formal specification for this problem.

```
 SYNCHRONIZER Low_n_high IS

 OPERATION CLASSES low,high;
 OPERATIONS lowop:low; highop:high;              (S1)

 RESOURCE STATE INFORMATION
  STATE VARIABLES                                (S2)
   maxavail CONSTANT 10
   inuse [0..maxavail] INITIALLY 0
  STATE CHANGES                                  (S3)
   Start(lowop) : inuse <- inuse+1
   Term(lowop)  : inuse <- inuse-1
  STATE INVARIANCE                               (S4)
   0 ≤ inuse  ≤ maxavail

 EXCLUSION         high's EXCLUDE                (S5)

 INTER CLASS PRIORITY AMONG REQUESTED OPERATIONS
          high > low                             (S6)

 SCHEDULING DISCIPLINE                           (S7)
 □{Req(highop) => ◇Start(highop)}
 □{[Req(lowop) &
        (~Req$high UNTIL Start(lowop))]
                          => ◇Start(lowop)}

 END low_n_high;
```

The following observations are pertinent:

- Instances of operations in a class can be referred to by using generic operation names, such as lowop and highop above.

- Normally, servicing constraint specifications express the conditions that should exist when an operation is serviced. In this example, the constraint is simply that a corresponding request be present. We have therefore omitted such constraints since they are implied by the axioms of the synchronization model.

- Data structures constituting the state of the resource, and modifications to the resource state by the operations, can be specified.

- There is a construct to specify invariance of a resource state predicate.

- Exclusion among operations belonging to the same class or different classes can be specified.

- Priority among operations within a class and between operations of different classes can be specified. In addition, priority can depend on resource state.

- Scheduling discipline statements specify the fairness that is expected of the synchronizer.

The specifications require that every high request be eventually serviced. Due to the presence of priority specifications, a weaker form of fairness is acceptable for low operations. Since requests are made by processes outside the synchronizer, the sequential model assumed precludes the immediate recognition of the presence of requests. This implies that, although at a given time a request may be eligible for service, a higher priority request may have arrived before the synchronizer recognizes this fact, thus preventing the synchronizer from servicing the former. Hence it is required that a low request eventually be serviced provided no high requests arrive before the low request would have been serviced (see L7 below).

Now we are in a position to give the semantics of the specifications for the problem in temporal logic. Below we have substituted formal temporal semantics Li for each specification statement Si. Init is a special condition which is true when the synchronizer is created and triggers its own negation.

```
init => (inuse=0)                                  (L2)
□(maxavail=10)

∀lowop∈low ∀p [Start(lowop) & inuse=p]             (L3)
                         TRIGGERS  inuse=p+1
∀lowop∈low ∀p [Term(lowop) & inuse=p]
                         TRIGGERS  inuse=p-1

□{0 ≤ inuse  ≤ maxavail}                           (L4)

∀p1,p2 ∈high p1≠p2,                                (L5)
         □~{Exec(p1) & Exec(p2)}

∀lowop∈low ∀highop∈high                            (L6)
 □{Req(lowop) & Req(highop)  =>
        Start(lowop) ONLYAFTER Start(highop)}

∀highop∈high                                       (L7)
□{Req(highop) => ◇Start(highop)}
∀lowop∈low
□{[Req(lowop) &
        (~Req$high UNTIL Start(lowop))]
                          => ◇Start(lowop)}
```

SENTINELS, A HIGH-LEVEL LANGUAGE CONSTRUCT FOR MULTIPROCESS COORDINATION

A sentinel is a special kind of process set up to provide tailored communication disciplines between other processes. The sentinel construct uses a queuing primitive as a basic form of synchronization. More elaborate forms of synchronization are then built up by constructing a sequential process (a sentinel) which coordinates other processes via the basic queuing primitive. The sentinel is the unique server of a set of queues which are associated with it. Sentinels allow a statement to be placed on the queue, in the sense that the sentinel can determine when that statement is to be executed, thus executing synchronization control over the enqueuing processes. Instead of requiring the synchronized processes to carry out certain clerical operations (e.g. causing other processes to be scheduled), a sentinel is an active process and carries out such operations itself.

In order to have a means for creating processes, we assume the underlying mechanics for a detached mode of execution, e.g., as with the "task" option in PL/I. For concreteness, we assume that any syntactic statement entity, <statement>, can be executed as a process by the statement

```
DETACH EXECUTE <statement>
```

which will create a process for <statement>, which then runs concurrently with the creating process. The statement which corresponds to a sentinel is a procedure call on the code of the sentinel. Queues are passed as parameters to that call. When the sentinel process is created, each queue is initialized. In order that all requests from enqueuing processes are enqueued, each queue has a "queue manager" which provides enqueuing processes exclusive access to it. The sentinel process created becomes the server of those queues.

The items which are communicated to sentinels from other processes via queues are called tokens. A token is a pair, consisting of a statement and a parameter list. Either of these items may be null in various applications. A token gets created by a process, called the enqueuing process, through a statement of the form

```
QUEUE(<queue ref>,<stmt>,<parameter list>).
```

The placement of a token puts the execution of <stmt> in control of a unique sentinel process serving the queue. It also makes any parameters in <parameter list> accessible to this server. The enqueuing process is suspended till the completion of execution of <stmt>.

The server services the statement component of the nth element in a queue by executing

```
EXECUTE <queue reference> [n]
```

where "n" is an integer variable whose value indicates the position from the front of the queue. The nth element is removed from the queue and cannot be re-executed. We also allow the detached mode of execution for an execute statement, viz.

```
DETACH EXECUTE <queue ref> [n] COUNT (c).
```

This effectively creates another temporary process which executes the statement part of the token in parallel with the synchronizer. Since the token is already a statement in another process, namely the enqueuing one, execution can be optimized so that no new process is actually created. The designated integer variable "c" will automatically be incremented by 1 when this statement is executed, and decremented by 1 when and if the detached process terminates.

We assume a wait until statement, which will delay a process until a specified condition becomes true. Empty(<queue reference>) tests whether the referred queue is empty and non-empty(<queue reference>), tests the negation. A sentinel recognizes that a request exists in a given queue Q only when it evaluates non-empty(Q) and finds it to be true.

The sentinel concept separates scheduling actions from the processes being scheduled. It should be mentioned that by demanding explicit selection of the next token to be executed, a sentinel does not provide internal nondeterminism as do similar independently-conceived mechanisms such as ADA[1] and serializers[2]. A user must program the sentinel to make a suitable choice. This allows for flexible, yet relatively easily-understood scheduling. (However, due to queueing delays global nondeterminism may be present in a distributed system.)

As an example, we give below the sentinel to synchronize low and high operations as specified in the previous section. (Statement labels are for subsequent reference.)

```
SENTINEL low_n_high (highq,lowq: QUEUE);
maxavail  CONSTANT INTEGER := 10;
inuse  INTEGER RANGE 0..max := 0;
La: WHILE true
     DO
Lb:  WAIT UNTIL (NON-EMPTY(highq) V
          (NON-EMPTY(lowq) & inuse < maxavail));
Lc:  WHILE NON-EMPTY(highq)
          DO
Ld:          EXECUTE  highq[1];
          OD
Le:  IF NON-EMPTY(lowq) & inuse < maxavail
     THEN
Lf:  DETACH EXECUTE lowq[1] COUNT (inuse)
     OD
END low_n_high;
```

This sentinel would be created by
DETACH EXECUTE low_n_high (highq,lowq).
A concurrent process requesting a high would execute
Queue (highq,highop,nil),
while a process requesting a low would execute
Queue (lowq,lowop,nil).

An implementation of a sentinel-like mechanism is described in [11].

FORMAL SEMANTICS OF SENTINEL CONSTRUCTS

The following predicates aid in referring to the instruction pointer of processes. For any executable statement S,

at(S)	is true iff control is at the beginning of S.
in(S)	is true iff control is within S.
after(S)	is true iff control is immediately following S.

For our purposes, the above informal definitions will suffice. Note that if S2 is the next executable statement following S1 then after(S1) is equivalent to at(S2).

1) By executing the statement S : "WAIT UNTIL C", the sentinel delays its own execution until the boolean condition C is true. Thus the semantics of "S" is

$$\{at(S) \Rightarrow [at(S)\ \mathrm{UNTIL}\ C]\}\ \&\ \Box\{[at(S)\ \&\ C] \Rightarrow \Diamond after(S)\}.$$

2) By executing the statement S: "QUEUE(Q, op, param)", a user process appends the token (op,param) to the queue named Q. Q[i] refers to the ith token in the queue. Q[i].op will refer to the operation component of the ith token. When there is no confusion as to whether the token or the operation component of the token is being referred to, we will use Q[i] to refer to the operation component. |Q| gives the number of tokens in the queue Q. Thus

$$\forall n\{[at(S)\ \&\ |Q|=n] \Rightarrow \Diamond[|Q|=n+1\ \&\ Q[n+1]=(op,param)\ \&\ Req(op)]\}.$$

3) By executing the statement S : "EXECUTE Q[n]", the sentinel process executes Q[n]. Let ^ denote concatenation of elements in queues and Q[n..m] denote a queue formed by elements Q[n], Q[n+1] ,..., Q[m-1], Q[m] of the queue "Q". The semantics of the execute statement is then:

$$\begin{aligned}&\forall Q1,Q2,p,n\\ &\{at(S)\ \&\ |Q|>n\ \&\ Q[n]=(op,\ param)\ \&\\ &\qquad p=|Q|\ \&\ \bar{Q}[1..n-1]=Q1\ \&\ Q[n+1..p]=Q2\}\\ &\Rightarrow\\ &\{Start(op)\ \&\\ &\ \Diamond[in(S)\ \&\ (Exec(op)\ \mathrm{UNTIL}\ after(S))\ \&\\ &\qquad\qquad Q[1..p-1]=Q1\hat{}Q2]\ \&\\ &\qquad\qquad \Box[after(S) \Rightarrow \sim Exec(op)]]\}\end{aligned}$$

4) By executing the statement S : "DETACH EXECUTE Q[n] COUNT(c)", the sentinel effectively makes an external task execute Q[n], i.e.,

$$\begin{aligned}&\forall\ Q1,Q2,k,m,n,p\\ &\{at(S)\ \&\ |Q|\geq n\ \&\ Q[n]=(op,\ param)\ \&\ c=m\ \&\\ &\qquad p=|Q|\ \&\ Q[1..n-1]=Q1\ \&\ Q[n+1..p]=Q2\}\\ &\Rightarrow\\ &\{Start(op)\ \&\\ &\quad \Diamond\{c=m+1\ \&\ Exec(op)\ \&\ Q[1..p-1]=Q1\hat{}Q2\}\ \&\\ &\qquad [Term(op)\ \&\ c=k]\ \mathrm{TRIGGERS}\ c=k-1\}\end{aligned}$$

From the above, we infer the following:

- The statement "QUEUE(Q,op,param)" enables the condition "Req(op)". Equivalently, $\forall op1\ \{\exists i\ Q[i].op=op1 \Rightarrow Req(op1)\}$. (I1)
- If S is (DETACH) EXECUTE Q[i] then $at(S) \Rightarrow Start(Q[i])$. (I2)
- Serving a request by means of a (DETACH) EXECUTE statement implies its eventual activation. Hence such a statement will eventually enable the condition "Exec(op)" where "op" is the serviced operation. (I3)

THE VERIFICATION TECHNIQUE

As mentioned earlier, there are two main phases in the verification process. In the first, conditions that hold when the synchronizer services an operation are determined. These are the constraints imposed by the sentinel for servicing an operation. Proof of these conditions is achieved by deriving place assertions[12] for the program from the semantics of the constructs involved, using the standard inductive assertions method[4, 7]. The preconditions of the "execute" statements give the conditions under which the operations are serviced.

In the second phase, from given high-level specifications, constraints are derived for the execution of an operation using the axioms and theorems of temporal logic. What then remains to be shown is: (1) given the preconditions for servicing operations, these constraints are satisfied, and (2) the sentinel guarantees the fairness specified.

To illustrate the proof technique, we will show that the sentinel which coordinates low and high operations is correct with respect to the specification of their synchronization. Given below is a list of the theorems of "linear time" version of temporal logic[14] that will be employed in the verification process. A and B denote arbitrary temporal logic expressions.

$$\begin{aligned}&T1 : \Box(A \vee B) \Rightarrow (\Box A \vee \Diamond B)\\ &T2 : \Diamond(A \vee B) \Leftrightarrow (\Diamond A \vee \Diamond B)\\ &T3 : \Box(A\ \&\ B) \Leftrightarrow (\Box A\ \&\ \Box B)\\ &T4 : \Box(A \Rightarrow \Diamond B)\ \&\ \Box(B \Rightarrow \Diamond C)\\ &\qquad\qquad \Rightarrow \Box(A \Rightarrow \Diamond C)\\ &T5 : (\Diamond A\ \&\ \Box B) \Rightarrow \Diamond(A\ \&\ B)\\ &T6 : \text{If } T \text{ is a theorem, then } \Box T\end{aligned}$$

We make the following assumption regarding terminating statements:

$$\Box\{[at(stmt)\ \&\ P] \Rightarrow \Diamond[after(stmt)\ \&\ P']\} \qquad (A1)$$

where P and P' are pre and post conditions appropriate for "stmt". This will be a valid assumption if the underlying scheduler of processes is fair.

Inductively, if S1,...,Sn is a sequence of terminating statements, then by A1, T4, and the definitions of the predicates "at" and "after", $at(S1) \Rightarrow \Diamond after(Sn)$.

We call this, "control flow reasoning".

Verification: Phase 1
From the semantics of the constructs, we determine assertions that hold at the beginning of certain statements (places) of interest. We assume that operations low and high always terminate. It then suffices to note that the place assertions below follow from the semantics of the statements and induction on the structure of the program. Here the statement labels correspond to those in the sentinel program.

```
at(Lb) => ~Exec$high
at(Lc) => ~Exec$high &
          (non-empty(highq) V
          (non-empty(lowq) & inuse<maxavail))
at(Ld) => ~Exec$high & non-empty(highq)
at(Lf) => ~Exec$high & inuse < maxavail & (b)
          EMPTY(highq) & non-empty(lowq)
```

Since only the sentinel can service requests, ~Exec$high and ~Exec$low are initially true.

For an operation "op" to be serviced, control should be at the beginning of an execute statement "S" that services "op". From the preconditions of the Execute statements, we have:

```
∀highop ∈ high
Start(highop) =>
  {non-empty(highq) & highop=highq[1] &
                          ~Exec$high}       (PR)

∀lowop ∈ low
Start(lowop) =>
 {non-empty(lowq) & lowop=lowq[1] &
    ~Exec$high & empty(highq) &
              inuse<maxavail)}         (PA)
```

Verification: Phase 2
We will systematically derive from the top level specifications the necessary conditions for servicing a request. These conditions are embedded in the scheduling constraint, mutual exclusion, resource state invariance, and priority specifications. We outline only the essential steps in deriving the necessary conditions from each of these statements. Only informal arguments are given for showing the correctness of the transformation steps. In what follows, to preserve readability, we have employed the following artifice. Statements in which the free variable "lowop" occurs are implicitly universally quantified over all low operations. Occurrences of "highop" are similarly quantified.

Transforming Mutual Exclusion Statement The specification statement "C excludes D", where C and D are operation classes, can be shown to be equivalent to

```
∀c ∈ C □{Start(c) =>  ~Exec$D}
∀d ∈ D □{Start(d) =>  ~Exec$C}
```

Exclusion of different instances of the same operation class, say C, is preserved if

```
∀c ∈ C □{Start(c) => ~Exec$C}
```

Thus the mutual exclusion of all high operations will be transformed into
  Start(highop) => ~Exec$high.

Translating Resource state invariance Invariance is maintained by ensuring that the Invariant is preserved by each operation that is serviced. For this purpose, a precondition is obtained for each operation from (a) the invariance specification and, (b) the changes to the resource state by the operation.

Translating Priority Specifications Given that high operations have higher priority than low operations, we have:

```
□{Req(lowop) & Req(highop) =>
   Start(lowop) ONLYAFTER Start(highop)}
```

This can be shown to be equivalent to

```
□{Start(lowop) => ~Req$high}
```

Conjunction of all constraints applicable to low and high operations respectively results in:

```
Start(lowop) =>
 {Req(lowop) & ~Req$high & inuse<max}     (CA)

Start(highop) =>
 {Req(highop) & ~Exec$High}               (CR)
```

Since

```
empty(highq)  => ~Req$high
highop=highq[1] => Req(highop)
lowop=lowq[1] => Req(lowop),
```

the preconditions PA and PR derived from the sentinel code, imply the constraints CA and CR respectively.

Observe that the sentinel overconstrains low operations since

```
Start(lowop) => ~Exec$high
```

from the sentinel code, whereas this is not required by the specifications. We can avoid such overconstraint by executing high operations also in a detached mode but avoid doing so for simplicity.

Proof of Fairness We reiterate our assumption that the operations being synchronized, namely low and high, always terminate. Before we prove the fairness specification, we prove three lemmas.

Lemma 1

```
□◇{at(Lc) V □at(Lb)}
```

(b) Control will be at Lf only when non-empty(highq) evaluates to false. Since the sentinel cannot recognize requests in highq until the next evaluation of non-empty(highq), empty(highq) is assumed to hold until then.

This lemma states that eventually control either remains forever at the WAIT UNTIL statement, or eventually reaches Lc.

Proof Given a statement w: WAIT UNTIL C, by T1,
□~C V <>C.

From the semantics of the WAIT UNTIL statement, and the definition of the UNTIL operator,
at(w) => □ at(w) V ◇after(w)

The Lemma then follows from control flow reasoning.

Lemma 2

∀C {C => ∀p □(p=highq[1] => ◇Start(p))} =>
  {C => ∀p□(∃i p=highq[i] => ◇Start(p))}

By this lemma, if we can show that the first request in highq will eventually be serviced, then all operations in highq will eventually be serviced.

Proof Assume the hypothesis of the lemma. From the semantics of the Execute statement,

∀i>1 ∀p {at(Ld) & highq[i]=p}
  => ◇{after(Ld) & highq[i-1]=p} (1)

Assume C and p=highq[i]. From the hypothesis of the lemma, and the semantics of the Execute statement, ◇Start(highq[1]).

◇Start(highq[1])

=> ◇at(Ld)
  -- definition of Start

=> ◇(p=highq[i-1])
  -- by (1)

By repeated application of (1) in conjunction with the hypothesis of the lemma, we get, ◇(p=highq[1]) and hence ◇Start(p).

Lemma 3

∀C {C => ∀p □(p=lowq[1] => ◇Start(p))} =>
  {C => ∀p □(∃i p=lowq[i] => ◇Start(p))}

This is similar to Lemma 2, and applies to low operations.

Proof of fairness to high requests
Fairness specified is

∀highop∈high □(Req(highop) =>
      ◇Start(highop)) (F1)

Let us consider one particular highop namely highq[1]. Using Lemma 1, either ◇□at(Lb) or ◇at(Lc). Since Req(highop), by the semantics of the Wait Until statement, □at(Lb) is false. If control is at Lc then since Req(highop), ◇at(Ld) which implies ◇Start(highop). So the first operation in highq will eventually be serviced, in which case, by lemma 2, all high operations will be eventually serviced. Now we give the formal proof.

We show that F1 is true for a particular highop, namely highq[1]. Suppose highop is never serviced. Then

Req(highop) =>
  [Req(highop) UNTIL Start(highop)]
  -- axiom of the synchronizer model

=> □Req(highop) -- definition of UNTIL. (2)

Furthermore,
◇□at(Lb) V ◇at(Lc) -- by Lemma 1 and T2

=> ◇[(at(Lb) & req(highop)) V at(Lc)]
  -- by T5, T2 and (2)

=> ◇at(Ld)
  -- using control flow reasoning and the semantics of "WAIT UNTIL" (3)

=> ◇Start(highop)

This contradicts the assumption □~Start(highop). Hence by T1, ◇Start(highop). Fairness to all high operations follows from Lemma 2.

Proof of Fairness to low requests
Fairness specified is

∀lowop∈low
 □{[Req(lowop) &
   (~Req$high UNTIL Start(lowop))]
      => <>Start(lowop)} (F2)

Here again we consider a particular lowop, namely lowq[1]. Assume that the request is never serviced. From the hypothesis of F2 and the definition of UNTIL, □Req(lowop) and □~Req$high are true. Since low operations terminate, ◇(inuse < maxavail). This conclusion in conjunction with lemma 1 and control flow reasoning shows that eventually control will reach Lf at which time lowop will be serviced. This contradicts the assumption and hence by T1, ◇Start(lowop).

Formally, assume □~Start(lowop). Since
Req(lowop) =>
  [Req(lowop) Until Start(lowop)],

from the semantics of UNTIL,

□Req(lowop) & □~Req$high. (4)

(inuse ≤ maxavail) is an invariant. (5)

inuse=maxavail

=> Exec$low

=> ∃lowop∈low Exec(lowop).

=> ◇[Term(lowop) & inuse=maxavail]

since all operations are assumed to terminate.

From the semantics of Detach Execute,

◇(inuse<maxavail). (6)

From T6 and (6),

$\Box\Diamond$(inuse < maxavail). (7)

From (4),(7) and control flow reasoning,

at(Lb) => $\Diamond$at(Lc)

=> $\Diamond$at(Lf) => $\Diamond$Start(lowop).

This contradicts our assumption $\Box$~Start(lowop).

F2 follows from T1 and Lemma 3.

To summarize, we have shown that (a) the conditions which hold when operations are serviced by the synchronizer conform to the constraints derived from the servicing constraints, invariance, priority and exclusion specifications, and (b) the synchronizer services operations according to the specified scheduling discipline, thus showing the correctness of the sentinel with respect to the overall specifications. We have given only the essential transformations from high-level specifications. Further description of the transformation system appears in[18].

CONCLUDING REMARKS

We have applied temporal logic to the specification and verification of synchronizers of concurrent processes. The full specification language[18] has constructs for stating the set of properties that are normally relevant to synchronization of concurrent operations, namely, ordering, fairness, priority, exclusion (and by default, concurrency), and invariance of resource state.

Our specification of semantics of the synchronizing constructs used in sentinels is complete in the sense that we have given the invariant and temporal behavior of each construct. Similarly, by specifying the behavior of primitives used to implement these constructs, we can prove their implementation.

Although we have applied the technique to sentinels, we conjecture that our technique is applicable to other synchronization mechanisms e.g., monitors[8], serializers[2] and ADA tasking facility[1]. The only requirement is that the synchronization primitives used must possess precise semantics. Then it will be possible to determine the conditions that should hold when operations are serviced, and thus apply the results of phase 2. One problem we do anticipate for these other mechanisms is in showing that a specified fairness is guaranteed by a synchronizer. This is due to the fact that some synchronizing constructs have "hidden" or unspecified scheduling disciplines (e.g. in ADA), or the underlying scheduling has one of many possible interpretations (e.g. in monitors[9]).

Previous efforts to specify synchronization include those of Griffiths[5] and Robinson[19]. In these, an abstract solution to the problem is specified, rather than the problem itself. Pre and post conditions are provided for "synchronizing functions" that surround critical regions. Verification entails showing that the code for the synchronizing functions behaves correctly with respect to the invariants specified.

Current techniques for verifying synchronized sharing of resources are mainly concerned with the proof of invariants. Keller[12] gave a general technique for this purpose. Owicki and Gries[15] presented proof rules using auxiliary variables. Hoare[8] gave proof rules for monitors which were later extended[10]. Path expressions[6] directly yield invariants which are used in the proof of processes synchronized by paths[3].

In all of the techniques mentioned in the previous paragraph, verification of invariants has been the main concern and proving correctness relies on the notion of states to reason indirectly about the effect of synchronization. We surmise that the uses of auxiliary (history) variables is mainly motivated by a need to maintain history information, i.e., temporal information. Further, formal proof of absence of deadlock and starvation requires techniques different from those used to prove invariants. On the other hand, the method we have outlined here handles invariant and temporal properties in a unified manner.

The approach used here for the proof suggests the possibility of synthesizing synchronizers from given specifications by extending phase 2. Preliminary ideas in this regard appear in[18]. Our experience with specifying and proving sentinels leads us to believe that temporal logic is a valuable tool in general, for a complete semantic definition of programming constructs, for invariant and temporal specifications of programs, and in formally verifying total correctness of these programs. By treating invariant and temporal properties under a single framework, temporal logic provides a unified approach to program verification and specification.

REFERENCES

[1] Ichbiah J.D. et al. Preliminary ADA Reference Manual. SIGPLAN Notices 14 (June 1979).

[2] Atkinson, R.R. and Hewitt, C.E. Specification and Proof Techniques for serializers. IEEE Transactions on Software Engineering SE-5 (Jan 1979), 10-23.

[3] Flon, L. and Habermann, A.N. Towards the Construction of Verifyable Software Systems. SIGPLAN Notices 11 (March 1976), 141-148.

[4] Floyd, R. Assigning Meanings to Programs. Proc. 9th. Symposium in Applied Mathematics, (1967).

[5] Griffiths, P. SYNVER: A System for the Automatic Synthesis and Verification of Synchronization Processes. TR 22-74, Harvard University, (1974).

[6] Habermann, A.N. Path Expressions. Carnegie-Mellon University, (June, 1975).

[7] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. Communications of the ACM 12 (1969), 576-580.

[8] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM 17 (Oct 1974), 540-557.

[9] Howard, J.H. Signalling in monitors. Proc. Second International Conference on Software Engineering, (Oct, 1976), pp. 47-52.

[10] Howard, J.H. Proving Monitors. Communications of the ACM 19 (May 1976), 549-557.

[11] B. Jayaraman and R.M. Keller. Resource control in a demand-driven data-flow model. Proc. 1980 International Conference on Parallel Processing, (August, 1980), pp. 118-127.

[12] Keller, R.M. Formal Verification of Parallel Programs. Communications of the ACM 19 (July 1976), 371-384.

[13] Keller, R.M. Sentinels: A Concept for Multiprocess Coordination. UUCS-78-104, University of Utah, (June, 1978).

[14] Lamport, L. 'Sometime' is Sometimes 'Not Never'. Proc. Seventh Annual Symposium on POPL, (Jan, 1980), pp. 174-185.

[15] Owicki, S. and Gries, D. Verifying Properties of Parallel Programs: An Axiomatic Approach. Communications of the ACM 19 (May 1976), 279-284.

[16] Owicki, S. Temporal logic and Parallel Programs. Colloquium, University of Utah.(Feb, 1979).

[17] Pnueli, A. The Temporal Semantics of Concurrent Programs. In Khan, Ed., Semantics of Concurrent Computation, Springer Lecture Notes in Computer Science, Springer-verlag, (1979), pp. 1-20.

[18] Ramamritham, K. and Keller, R.M. Specification and Synthesis of Synchronizers. Proc. 1980 International Conference on Parallel Processing, (Aug, 1980).

[19] Robinson, L. and Holt, R.C. Formal Specifications for Solutions to Synchronization Problems. Stanford Research Institute, (1975).