KRML 41 / HPH 23 -0

Semaphore specifications: Larch meets Martin

H. Peter Hofstee, K. Rustan M. Leino, Jan L.A. van de Snepscheut
California Institute of Technology
18 August 1994

We present a Larch [2] specification of semaphores that satisfies Alain Martin's semaphore axioms [6]. As a result, the semaphore specification becomes strong enough to enable the implementation of fair semaphores from unfair ones. As Larch cannot express fairness, this is the best semaphore specification we can hope for in Larch.

0 Introduction

The information contained in this note is a late write-up of a group meeting at Caltech during the fall of 1992. Since then, the last-mentioned author passed away; hence, any mistakes in this presentation are solely due to the other authors.

The group meeting started with an expressed disappointment in the Larch specification of procedures *Acquire* and *Release* in [0].

```
**private var** $holder : \mathbf{array}\ Mutex\ \mathbf{of}\ T$
**initially** $\langle \forall\, m : Mutex \triangleright\ holder[m] = \mathbf{nil}\ \rangle$

**procedure** $Acquire(m : Mutex)$;
  **modifies** $holder[m]$
  **when** $holder[m] = \mathbf{nil}$
  **ensures** $holder'[m] = current$

**procedure** $Release(m : Mutex)$;
  **requires** $holder[m] = current$
  **modifies** $holder[m]$
  **ensures** $holder'[m] = \mathbf{nil}$
```

Type T signifies $Thread.T$, the type representing threads (light-weight processes) in Modula-3. The special value *current* refers to the value of the calling process. No process takes the value **nil**. Procedure *Acquire* on a mutex m is like a P operation on a semaphore [1], and *Release* is like a V operation but with the restriction that the process that executes *Release* must be the "holder" of the lock. In Larch, primes indicate the post-value of variables.

As is usual, we assume a process that acquires a lock on a semaphore will eventually release it. Then, a semaphore is said to be *fair* if no process is suspended on a P operation forever. A semaphore may or may not be fair, but fair semaphores can be implemented using three unfair ones [7], or two unfair ones

[3]. (In our opinion, [4] and [5] are improvements on the proof of [3].) The correctness of these algorithms depends on the following axioms [6]:

$$\begin{array}{ll} A0: & 0 \le s \le 1 \\ A1: & \mathbf{c}P + s = \mathbf{c}V + s_0 \\ A2: & \mathbf{q}P = 0 \lor s = 0 \qquad , \end{array}$$

where s is the integer value of the semaphore, the initial value of which is s_0, $\mathbf{c}OP$ is the number of completed OP operations, and $\mathbf{q}OP$ is the number of suspended OP operations. Actually, Martin uses only $0 \le s$ for $A0$, but we have added the other bound to show that the semaphores are binary. Note that V operations never suspend.

Our disappointment with the specification in [0] arose from that it is not strong enough to prove Martin's axioms. Therefore, one cannot build fair semaphores from the specified ones. [0] also includes specifications for procedures P and V, but those are even weaker than the ones for *Acquire* and *Release*. We start our hunt for a stronger specification.

1 Larch specification

We give our Larch specifications for procedures P and V.

```
private var $holder : T$
            $q :$ set of $T$
            $i :$ integer
initially $holder = \mathbf{nil} \land q = \{\} \land i = 1$

procedure $P()$;
  modifies $holder, q, i$
  composition of $Queue; Dequeue$ end
  action $Queue$
    requires $holder \neq current$
    ensures $(holder = \mathbf{nil} \land holder' = current \land i' = i - 1 \land \mathbf{unchanged}(q)) \lor$
            $(holder \neq \mathbf{nil} \land q' = q \cup \{current\} \land \mathbf{unchanged}(holder, i))$
  action $Dequeue$
    when $current = holder$
    ensures $\mathbf{unchanged}(holder, q, i)$

procedure $V()$;
  modifies $holder, q, i$
  requires $holder = current$
  ensures $(q \neq \{\} \land holder' \in q \land q' = q \setminus \{holder'\} \land \mathbf{unchanged}(i)) \lor$
          $(q = \{\} \land holder' = \mathbf{nil} \land i' = i + 1 \land \mathbf{unchanged}(q))$
```

A procedure is either one atomic action or a composition of atomic actions. In the above, V is atomic whereas P is a composition of the two atomic actions $P.Queue$ and $P.Dequeue$. $\mathbf{unchanged}(w)$ is a shorthand for $w' = w$.

Given the above specification, we prove the following invariants:

$J0: \ \mathbf{nil} \notin q$
$J1: \ holder \neq \mathbf{nil} \vee q = \{\}$
$J2: \ holder \notin q$
$J3: \ (holder = \mathbf{nil} \equiv i = 1) \wedge (holder \neq \mathbf{nil} \equiv i = 0)$

1.0 Proof of J0

$J0$ holds initially, and the only element added to q is $current$, which is not $\mathbf{nil}$.

1.1 Proof of J1

Initially, $q = \{\}$, from which the proof obligation directly follows.
For $P.Queue$,

$$\begin{array}{ll} & J1 \ \wedge \ P.Queue.Requires \ \wedge \ P.Queue.Ensures \\ \Rightarrow & \quad \{ \quad holder \neq \mathbf{nil} \vee holder = \mathbf{nil} \quad \} \\ & (holder \neq \mathbf{nil} \wedge holder' = holder) \ \vee \ (holder = \mathbf{nil} \wedge q = \{\} \ \wedge \ q' = q) \\ \Rightarrow & \\ & holder' \neq \mathbf{nil} \vee q' = \{\} \qquad . \end{array}$$

No variables are changed by $P.Dequeue$.
For V,

$$\begin{array}{ll} & J1 \ \wedge \ V.Requires \ \wedge \ V.Ensures \\ \Rightarrow & \\ & holder' \in q \ \vee \ (q = \{\} \ \wedge \ q' = q) \\ \Rightarrow & \quad \{ \quad J0 \quad \} \\ & holder' \neq \mathbf{nil} \vee q' = \{\} \qquad . \end{array}$$

1.2 Proof of J2

Initially, $q = \{\}$, from which the proof obligation directly follows.
For $P.Queue$,

$$\begin{array}{ll} & J2 \ \wedge \ P.Queue.Requires \ \wedge \ P.Queue.Ensures \\ \Rightarrow & \quad \{ \quad J1 \quad \} \\ & (holder = \mathbf{nil} \wedge q' = q \ \wedge \ q = \{\}) \vee \\ & (q' = q \cup \{current\} \ \wedge \ holder' \notin q \ \wedge \ holder' = holder \ \wedge \ holder \neq current) \\ \Rightarrow & \\ & holder' \notin q' \qquad . \end{array}$$

No variables are changed by $P.Dequeue$.
For V,

$$
\begin{array}{ll}
 & J2 \;\wedge\; V.Requires \;\wedge\; V.Ensures \\
\Rightarrow & \\
 & q' = q \setminus \{holder'\} \;\vee\; holder' = \mathbf{nil} \\
\Rightarrow & \quad \{ \quad J0 : \mathbf{nil} \notin q' \quad \} \\
 & holder' \notin q' \qquad .
\end{array}
$$

1.3 Proof of J3

Initially, $holder = \mathbf{nil} \wedge i = 1$, from which the proof obligation directly follows.

For $P.Queue$,

$$
\begin{array}{ll}
 & J3 \;\wedge\; P.Queue.Requires \;\wedge\; P.Queue.Ensures \\
\Rightarrow & \\
 & (holder = \mathbf{nil} \wedge i = 1 \;\wedge\; holder' = current \;\wedge\; i' = i - 1) \vee \\
 & (holder \neq \mathbf{nil} \wedge i = 0 \;\wedge\; holder' = holder \;\wedge\; i' = i) \\
\Rightarrow & \\
 & holder' \neq \mathbf{nil} \wedge i' = 0 \\
\Rightarrow & \\
 & (holder' = \mathbf{nil} \;\equiv\; i' = 1) \;\wedge\; (holder' \neq \mathbf{nil} \;\equiv\; i' = 0) \qquad .
\end{array}
$$

No variables are changed by $P.Dequeue$.

For V ,

$$
\begin{array}{ll}
 & J3 \;\wedge\; V.Requires \;\wedge\; V.Ensures \\
\Rightarrow & \\
 & holder = current \;\wedge\; holder \neq \mathbf{nil} \wedge i = 0 \;\wedge \\
 & ((q \neq \{\} \;\wedge\; holder' \in q \;\wedge\; i' = i) \;\vee\; (holder' = \mathbf{nil} \wedge i' = i + 1)) \\
\Rightarrow & \quad \{ \quad holder' \neq \mathbf{nil} \vee holder' = \mathbf{nil} \quad \} \\
 & (holder' \neq \mathbf{nil} \wedge i' = 0) \;\vee\; (holder' = \mathbf{nil} \wedge i' = 1) \\
\Rightarrow & \\
 & (holder' = \mathbf{nil} \;\equiv\; i' = 1) \;\wedge\; (holder' \neq \mathbf{nil} \;\equiv\; i' = 0) \qquad .
\end{array}
$$

2 Martin's axioms

In order to show that our Larch specification satisfies Martin's axioms, we need to discuss s , s_0 , $\mathbf{c}V$, $\mathbf{c}P$, and $\mathbf{q}P$. We claim s corresponds to i , and $s_0 = 1$. Consequently, $A0$ follows directly from $J3$.

The other values model how many times the different atomic actions have been executed. Thus, every completion of action $P.Queue$ ought to result in an increment of $\mathbf{q}P$, every completion of $P.Dequeue$ in a decrement of $\mathbf{q}P$ and an increment in $\mathbf{c}P$, and every completion of V in an increment of $\mathbf{c}V$. We will refer to these rules as the *intended meanings* of $\mathbf{q}P$, $\mathbf{c}P$, and $\mathbf{c}V$. Note that these values are non-negative integers, and that the latter two are only increased.

We add variables

private var cV, cP, qP : **integer**
initially $cV = 0 \wedge cP = 0 \wedge qP = 0$,

which to correspond to $\mathbf{c}V$, $\mathbf{c}P$, and $\mathbf{q}P$, respectively. In planting the updates of these variables, we will consult the intended meanings as well as Martin's axioms. We will not change the behavior of the previously introduced variables.

Guided by the axioms, we start by sinking our teeth into $A1$. Variable i is modified in the first disjunct of $P.Queue.Ensures$ and in the second disjunct of $V.Ensures$. To comply with $A1$, we will compensate for these modifications by adding an increase of cP to the former and an increase of cV to the latter.

To maintain $A2$, we need to consider increases of i (and, due to $A0$, these are the only things to worry about). As q is empty at the time i is increased in $V.Ensures$, we let qP correspond to the size of q. Consequently, we add an increment of qP to the second disjunct of $P.Queue.Ensures$ and a decrement of qP to the first disjunct of $V.Ensures$. As $A0$ and $A1$ don't mention qP, they are still maintained, and since qP is invariably the size of q, it is always non-negative.

Now that the axioms are satisfied, we consider the intended meanings while maintaining the invariance of the axioms. Variable cV is supposed to correspond to the number of completed V operations. This propels us to add an increase of cV to the first disjunct of $V.Ensures$ (recall that we already extended the second disjunct with one). As we have given up additional modifications of i, a simultaneous increase of cP is called for, in order to maintain $A1$. We have now arrived upon the specification

```
private var $holder : T$
            $q : \textbf{set of } T$
            $i, cV, cP, qP : \textbf{integer}$
initially $holder = \textbf{nil} \wedge q = \{\} \wedge i = 1 \wedge cV = 0 \wedge cP = 0 \wedge qP = 0$

procedure $P()$;
  modifies $holder, q, i, cP, qP$
  composition of $Queue; Dequeue$ end
  action $Queue$
    requires $holder \neq current$
    ensures $(holder = \textbf{nil} \wedge holder' = current \wedge i' = i - 1 \wedge$
                $cP' = cP + 1 \wedge \textbf{unchanged}(q, qP)) \vee$
            $(holder \neq \textbf{nil} \wedge q' = q \cup \{current\} \wedge$
                $qP' = qP + 1 \wedge \textbf{unchanged}(holder, i, cP))$
  action $Dequeue$
    when $current = holder$
    ensures $\textbf{unchanged}(holder, q, i, cP, qP)$

procedure $V()$;
  modifies $holder, q, i, cV, cP, qP$
  requires $holder = current$
  ensures $(q \neq \{\} \wedge holder' \in q \wedge q' = q \setminus \{holder'\} \wedge$
              $qP' = qP - 1 \wedge cP' = cP + 1 \wedge cV' = cV + 1 \wedge \textbf{unchanged}(i)) \vee$
          $(q = \{\} \wedge holder' = \textbf{nil} \wedge i' = i + 1 \wedge$
              $cV' = cV + 1 \wedge \textbf{unchanged}(q, qP, cP))$ .
```

So what about the intended meanings of qP and cP? We see that the update

$$cP' = cP + 1$$

occurs in $P.Queue.Ensures$ instead of in $P.Dequeue.Ensures$, and the updates

$$qP' = qP - 1 \wedge cP' = cP + 1$$

appear in $V.Ensures$ instead of in $P.Dequeue.Ensures$. The crux is that the axioms dictate that sometimes a pair of P and V operations terminate at the same time.

So is our specification still good? The answer will come from reexamining the "intended meanings". We will take the stand that an operation completes when all remaining **when** clauses of its execution are *true* and are *stable*, that is, they will remain *true* until the action has completed. An operation is suspended if it has started but has not yet completed.

With this interpretation in mind, we need to show that $current = holder$ holds as a result of the first disjunct of $P.Queue.Ensures$, and that this condition is stable. We also need to show that the value of $holder$ resulting from the first disjunct of $V.Ensures$ is stable.

The first disjunct of $P.Queue.Ensures$ implies $current = holder'$, so $current = holder$ will hold after the action. There is only one process corresponding with this value of $holder$, so no other process can satisfy the precondition of V. Moreover, any other process that embarks on a P operation will find $holder \neq \textbf{nil}$, so it will not change the value of $holder$. We conclude our proof obligation holds for the first disjunct of $P.Queue.Ensures$.

The first disjunct of $V.Ensures$ mandates $holder' \in q$, so upon completion of V, $holder$ will equal some non-**nil** value, call it t. Processes other than t can then not meet the precondition of V, and any process embarking on a P operation will find $holder \neq \textbf{nil}$, and thus end up not altering the value of $holder$. This concludes our proof.

In conclusion, with a less strict interpretation of the rôle of the variables, we have shown that our specification satisfies the axioms.

3 Multiple semaphores

It is easy to extend our specification to cater for multiple semaphores, because the specification of each semaphore shares no variables between the specifications of other semaphores.

```
private var $holder$ : array $Mutex$ of $T$
            $q$ : array $Mutex$ of set of $T$
            $i$ : array $Mutex$ of integer
initially $\langle \forall\, m : Mutex \triangleright\ holder[m] = \textbf{nil} \wedge q[m] = \{\} \wedge i[m] = 1 \,\rangle$
```

```
procedure $P(m : Mutex)$;
  modifies $holder[m], q[m], i[m]$
  composition of $Queue; Dequeue$ end
  action $Queue$
    requires $holder[m] \neq current$
    ensures $(holder[m] = \mathbf{nil} \wedge holder'[m] = current \wedge i'[m] = i[m] - 1 \wedge$
                 $\mathbf{unchanged}(q[m])) \vee$
            $(holder[m] \neq \mathbf{nil} \wedge q'[m] = q[m] \cup \{current\} \wedge$
                 $\mathbf{unchanged}(holder[m], i[m]))$
  action $Dequeue$
    when $current = holder[m]$
    ensures $\mathbf{unchanged}(holder[m], q[m], i[m])$

procedure $V(m : Mutex)$;
  modifies $holder[m], q[m], i[m]$
  requires $holder[m] = current$
  ensures $(q[m] \neq \{\} \wedge holder'[m] \in q[m] \wedge q'[m] = q[m] \setminus \{holder'[m]\} \wedge$
               $\mathbf{unchanged}(i[m])) \vee$
          $(q[m] = \{\} \wedge holder'[m] = \mathbf{nil} \wedge i'[m] = i[m] + 1 \wedge$
               $\mathbf{unchanged}(q[m]))$ .
```

4 Conclusions

To write a semaphore specification strong enough to satisfy Martin's axioms, we split the specification of P into two atomic actions. We provided an interpretation for $\mathbf{c}P$ and $\mathbf{q}P$, and proved that the specification satisfies the axioms.

5 Acknowledgements

We are grateful for discussions with Greg Nelson during the fall of 1992.

References

[0] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. Thread synchronization: A formal specification. In Greg Nelson, editor, *Systems Programming with Modula-3*, Prentice Hall Series in Innovative Technology, pages 119–129. Prentice Hall, 1991.

[1] E.W. Dijkstra. The structure of the 'THE'—multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[2] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[3] S. Haldar and D.K. Subramanian. A fair solution to the mutual exclusion problem using weak semaphores. *Operating Systems Review*, 22(1):60–66, April 1988.

[4] S. Haldar, D.K. Subramanian, and D. Gries. One-bounded mutual exclusion using two blocked-set binary semaphores and two shared bits. Private communications, 1991.

[5] H.P. Hofstee, K.R.M. Leino, and J.L.A. van de Snepscheut. Proof of a mutual exclusion algorithm by Haldar and Subramanian. HPH 11, Internal note, California Institute of Technology, December 1991.

[6] A.J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[7] A.J. Martin and J.R. Burch. Fair mutual exclusion with unfair P-operations and V-operations. *Information Processing Letters*, 21(2):97–100, 1985.