6.10 Barz's simulation of general semaphores[A]

In this section we present Hans W. Barz's simulation of a general semaphore by a pair of binary semaphores and an integer variable [6]. (See [67] for a survey of the attempts to solve this problem.) The simulation will be presented within the

Algorithm 6.13: Barz's algorithm for simulating general semaphores

```
                    binary semaphore S ← 1
                    binary semaphore gate ← 1
                    integer count ← k

     loop forever
        non-critical section

p1:     wait(gate)                     // Simulated wait
p2:     wait(S)
p3:     count ← count − 1
p4:     if count > 0 then
p5:        signal(gate)
p6:     signal(S)

        critical section

p7:     wait(S)                        // Simulated signal
p8:     count ← count + 1
p9:     if count = 1 then
p10:       signal(gate)
p11:    signal(S)
```

context of a solution of the critical section problem that allows $k > 0$ processes simultaneously in the critical section (Algorithm 6.13).

k is the initial value of the general semaphore, statements p1..6 simulate the wait statement, and statements p7..11 simulate the signal statement. The binary semaphore gate is used to block and unblock processes, while the variable count holds the value of the integer component of the simulated general semaphore. The second binary semaphore S is used to ensure mutual exclusion when accessing count. Since we will be proving only the safety property that mutual exclusion holds, it will be convenient to consider the binary semaphores as busy-wait semaphores, and to write the value of the integer component of the semaphore gate as *gate*, rather than *gate.V*.

It is clear from the structure of the algorithm that S is a binary semaphore; it is less clear that gate is one. Therefore, we will not assume anything about the value of gate other than that it is an integer and prove (Lemma 6.5(6), below) that it only takes on the values 0 and 1.

Let us start with an informal description of the algorithm. Since gate is initialized to 1, the first process attempting to execute a simulated wait statement will succeed in passing p1: wait(gate), but additional processes will block. The first process and each subsequent process, up to a total of $k-1$, will execute p5: signal(gate), releasing additional processes to successfully complete p1: wait(gate). The if statement at p4 prevents the kth process from executing p5: signal(gate), so further processes will be blocked at p1: wait(gate).

When $count = 0$, a single simulated signal operation will increment count and execute p11: signal(gate), unblocking one of the processes blocked at p1: wait(gate); this process will promptly decrement count back to zero. A sequence of simulated signal operations, however, will cause count to have a positive value, although the value of *gate* remains 1 since it is only signaled once. Once count has a positive value, one or more processes can now successfully execute the simulated wait.

An inductive proof of the algorithm is quite complex, but it is worth studying because so many incorrect algorithms have been proposed for this problem.

In the proof, we will reduce the number of steps in the induction to three. The binary semaphore S prevents the statements p2..6 from interleaving with the statements p7..11. p1 can interleave with statements p2..6 or p7..11, but cannot affect their execution, so the effect is the same as if *all* the statements p2..6 or p7..11 were executed before p1.[3]

We will use the notation *entering* for *p*2..6 and *inCS* for *p*7. We also denote by *#entering*, respectively *#inCS*, the number of processes for which *entering*, respectively *inCS*, is true.

Lemma 6.5 The conjunction of the following formulas is invariant:

$$
\begin{array}{ll}
(1) & entering \rightarrow (gate = 0), \\
(2) & entering \rightarrow (count > 0), \\
(3) & \#entering \leq 1, \\
(4) & ((gate = 0) \wedge \neg\, entering) \rightarrow (count = 0), \\
(5) & (count \leq 0) \rightarrow (gate = 0), \\
(6) & (gate = 0) \vee (gate = 1).
\end{array}
$$

[3]Formally, wait statements are *right movers* [46].

Proof: The phrase "by (n), A holds" will mean: by the inductive hypothesis on formula (n), A must be true before executing this statement. The presentation is somewhat terse, so before reading further, make sure that you understand how material implications can be falsified (Appendix B.3).

Initially:
(1) All processes start at $p1$, so the antecedent is false.
(2) As for (1).
(3) $\#entering = 0$.
(4) $gate = 1$ so the antecedent is false.
(5) $count = k > 0$ so the antecedent is false.
(6) $gate = 1$ so the formula is true.

Executing p1:
(1) By (6), $gate \le 1$, so p1: wait(gate) can successfully execute only if $gate = 1$, making the consequent $gate = 0$ true.
(2) *entering* becomes true, so the formula can be falsified only if the consequent is false because $count \le 0$. But p1 does not change the value of count, so we must assume $count \le 0$ before executing the statement. By (5), $gate = 0$, so the p1: wait(gate) cannot be executed.
(3) This can be falsified only if *entering* is true before executing p1. By (1), if *entering* is true, then $gate = 0$, so the p1: wait(gate) cannot be executed.
(4) *entering* becomes true, so the antecedent $\neg$ *entering* becomes false.
(5) As for (1).
(6) As for (1).

Executing p2..6:
(1) Some process must be at p2 to execute this statement, so *entering* is true, and by (3), $\#entering = 1$. Therefore, the antecedent *entering* becomes false.
(2) As for (1).
(3) As for (1).
(4) By (1), $gate = 0$, and by (2), $count > 0$. If $count = 1$, the consequent $count = 0$ becomes true. If $count > 1$, p3, p4 and p5 will be executed, so that *gate* becomes 1, falsifying the antecedent.
(5) By (1), $gate = 0$, and by (2), $count > 0$. If $count > 1$, after decrementing its value in p3, $count > 0$ will become true, falsifying the antecedent. If $count = 1$, the antecedent becomes true, but p5 will not be executed, so $gate = 0$ remains true.
(6) By (1), $gate = 0$ and it can be incremented only once, by p5.

Executing p7..11:
(1) The value of *entering* does not change. If it was true, by (2) $count > 0$, so that *count* becomes greater than 1 by p8, ensuring by the if statement at p9 that the value of *gate* does not change in p10.

(2) The value of *entering* does not change, and the value of *count* can only increase.
(3) Trivially, the value of *#entering* is not changed.
(4) Suppose that the consequent $count = 0$ were true before executing the statements and that it becomes false. By the if statement at p9, statement p10: signal(gate) is executed, so the antecedent $gate = 0$ is also falsified. Suppose now that both the antecedent and the consequent were false; then the antecedent cannot become true, because the value of *entering* does not change, and if $gate = 0$ is false, it certainly cannot become true by executing p10: signal(gate).
(5) The consequent can be falsified only if $gate = 0$ were true before executing the statements and p10 was executed. By the if statement at p9, that can happen only if $count = 0$. Then $count = 1$ after executing the statements, falsifying the antecedent. If the antecedent were false so that $count > 0$, it certainly remains false after incrementing count in p8.
(6) The formula can be falsified only if $gate = 1$ before executing the statements and p10: signal(gate) is executed. But that happens only if $count = 0$, which implies $gate = 0$ by (5). ∎

Lemma 6.6 The formula $count = k - \#inCS$ is invariant.

Proof: The formula is initially true by the initialization of count and the fact that all processes are initially at their non-critical sections. Executing p1 does not change any term in the formula. Executing p2..6 increments *#inCS* and decrements *count*, preserving the invariant, as does executing p7..11, which decrements *#inCS* and increments *count*. ∎

Theorem 6.7 Mutual exclusion holds for Algorithm 6.13, that is, $\#inCS \leq k$ is invariant.

Proof: Initially, $\#inCS \leq k$ is true since $k > 0$. The only step that can falsify the formula is p2..6 executed in a state in which $\#inCS = k$. By Lemma 6.6, in this state $count = 0$, but *entering* is also true in that state, contradicting (2) of Lemma 6.5. ∎

The implementation of Barz's algorithm in Promela is discussed in Section 6.15.