15-398, Fall 2002
Final Project

Due December 5, 2002

All final projects, including this one, are due the last day of the classes, i.e., December 5, 2002.

This is a project that every one should do if they haven't found out a topic for project yet. You should **not collaborate** with others on this project. Submit your own hand-in.

1 The Cigarette Smoker's Problem

We will model an interesting synchronization problem, known as the cigarette smoker's problem in SMV or Promela/SPIN. This problem is due to S. S. Patil in 1971 ([1]). Suppose a cigarette requires three ingredients, tobacco, paper and match. There are three chain smokers (resource users). Each of them has only one ingredient with infinite supply. There is an agent who has infinite supply of all three ingredients. To make a cigarette, the smoker has tobacco (resp., paper and match) must have the other two ingredients paper and match (resp., tobacco and match, and tobacco and paper). The agent and smokers share a table. The agent randomly generates two ingredients and notifies the smoker who needs these two ingredients. Once the ingredients are taken from the table, the agent supplies another two. On the other hand, each smoker waits for the agent's notification. Once it is notified, the smoker picks up the ingredients, makes a cigarette, smokes for a while, and goes back to the table waiting for his next ingredients. The problem is to come up with an algorithm for the smokers using semaphores as synchronization primitives. Semaphores are described in section 2. The algorithm that an agent follows is given in Figure 1 as a collection of six processes $r_a, r_b, r_c, \beta_x, \beta_y, \beta_z$.

semaphore s; (initially 1) semaphore a, b, c, X, Y, Z; (all initially 0)		
r_a : $P(s)$; $V(b)$; $V(c)$; **go to** r_a;	r_b : $P(s)$; $V(a)$; $V(c)$; **go to** r_b;	r_c : $P(s)$; $V(a)$; $V(b)$; **go to** r_c;
β_x : $P(X)$; $V(s)$; **go to** β_x;	β_y : $P(Y)$; $V(s)$; **go to** β_y;	β_z : $P(Z)$; $V(s)$; **go to** β_z;

Figure 1: Six processes algorithm for the agent in the Cigarette Smoker's Problem.

The semaphores a, b and c are used by the agent to report the availability of the three resources. Each semaphore denotes one of the resources. s is a mutual exclusion semaphore to assure that only one pair of

resources is made available at a time. X, Y, Z are to be used by the resource users to report that they are done with the resources.

Patil contended that there is no solution to the problem without using conditional statements and without using simultaneous semaphore primitives and semaphore arrays. However, Parnas in [2] showed that the restriction of not having a semaphore array was unreasonable and presented a simple solution using an array of semaphores without any conditional statement or simultaneous semaphore operation. This solution is presented below in Figure 2 with nine processes $\delta_a, \delta_b, \delta_c, \alpha_x, \alpha_y, \alpha_z$ and semaphore overflow prevention processes $\delta_1, \delta_2, \delta_3$. For full description of the solution, refer to [2].

semaphore $mutex$; (initially 1) integer t; (initially 0) semaphore array $S[1..6]$; (all initially 0)		
δ_a : $P(a)$; $P(mutex)$; $t \leftarrow t + 1$; $V(S[t])$; $V(mutex)$; **go to** δ_a;	δ_b : $P(b)$; $P(mutex)$; $t \leftarrow t + 2$; $V(S[t])$; $V(mutex)$; **go to** δ_b;	δ_c : $P(c)$; $P(mutex)$; $t \leftarrow t + 4$; $V(S[t])$; $V(mutex)$; **go to** δ_c;
α_x : $P(S[6])$; $t \leftarrow 0$; // smoking now $V(X)$; **go to** α_x;	α_y : $P(S[5])$; $t \leftarrow 0$; // smoking now $V(Y)$; **go to** α_y;	α_z : $P(S[3])$; $t \leftarrow 0$; // smoking now $V(Z)$; **go to** α_z;
δ_1 : $P(S[1])$; **go to** δ_1;	δ_2 : $P(S[2])$; **go to** δ_2;	δ_3 : $P(S[4])$; **go to** δ_3;

Figure 2: Nine processes algorithm for three resource users in the Cigarette Smoker's Problem. All semaphores, variables and processes defined here are in addition to the agent of Figure 1.

We will be implementing this solution in SMV or Promela/SPIN and verify some interesting properties of the resource users.

2 Semaphores

Semaphores were introduces by Dijkstra in [3] in 1965. They are used for synchronization between processes. A semaphore is a shared variable that supports to operations P or `down` and V or `up`. If a process wants to do a P operation on a semaphore s, and $s > 0$, then the value of s is atomically decremented and the processes goes ahead. On the other hand, if $s = 0$, then the process blocks until s becomes greater than 0. A semaphore is atomically incremented by a V operation. A process never blocks while doing V operations on a semaphore. Typically, there is a bound on the size of the semaphore. Once the value of the semaphore reaches that bound, further V operations on the semaphore do not have any effect. The value of the semaphore stays the same. Note that it *does not* roll-over to 0 as with other bounded variables in a programming language. These semaphores are known as counting semaphores. If a semaphore counts up to 1 only, then it is called a binary semaphore.

3 Modeling in SMV or Promela/SPIN

You should begin by first modeling a working semaphore. Any modern OS typically provides an implementation of semaphores. You should model a semaphore in SMV by implementing mutual exclusion algorithm for *asynchronous systems*. Note that all processes that may be using a semaphore need to participate in the mutual exclusion, so it is not a two way mutual exclusion. Therefore, you will have to think of something like Peterson's algorithm for mutual exclusion. Another option is to implement mutual exclusion by a request-grant signal pair for each process that is participating in the mutual exclusion. In this case, a non deterministic variable would decide which process gets access to critical region. Another thing to note is that you will have to implement a semaphore with a finite size. Therefore, processes δ_1, δ_2 and δ_3 are required. Given these processes, a bound of 1 (binary semaphore) should suffice. However, you will write a property to verify that the bound is indeed not exceeded given these three processes.

The property that you should verify for a semaphore is that once the semaphore reaches 0, it should not allow another P operation until a V operation is done on it. If you implemented mutual exclusion, you will also want to verify the usual mutual exclusion properties, i.e, **safety** and **liveness**.

Once you've got a semaphore working, then following the algorithms for agent and the users should be easy. You should specify the running fairness constraint to make sure each process gets a chance to run infinitely often. Also note that if you doing this in SMV, you will have to implement a state machine for each process that essentially captures the sequential flow of the statements. The simplest way to do this to label each statement by a program counter or a state number and following the sequential flow for defining the transition relation. Promela allows you to do this easily however, since it provides for sequential composition.

Once you have these implemented, you should state following property and verify them.

Liveness The smokers do get to smoke infinitely often, i.e., they visit the section marked with the comment `// smoking now` infinitely often.

Safety No two smokers are smoking at the same time.

No Strict Sequencing You will state the pairwise no strict sequencing property.

4 Submission

1. A detailed report describing your implementation of semaphores, agent and the user processes, CTL/LTL properties and verification results (including running time, memory requirement).
2. SMV/Promela programs for semaphores, agents and user processes.
3. Output of the model checkers clearly showing that the properties were verified.

You should feel free to experiment with your own ideas for solving this problem and may want to refer to the original paper [1]. The original paper describes a solution with conditional statements and simultaneous semaphore operation. You may want to experiment with that solution.

References

[1] Patil, S.S. Limitations and capabilities of Dijkstra's semaphore primitives for co-ordination among processes. Proj. MAC, Computational Structures Group Memo 57. Feb. 1971.

[2] Parnas, D.L. On a solution to the Cigarette smoker's problem (without conditional statements). Communications of the ACM. pp. 181–183. Volume 18, Number 3, March 1975.

[3] Dijkstra, E.W., 1965. Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9):569, 1965.