Timed and Hybrid Systems
Spring 2007: Assignment No. 1

Due Date: 2.26.07
February 12, 2007

This assignment requires extensive use of the TLV verifier. Please refer to the TLV web page http://www.cs.nyu.edu/acsys/tlv for obtaining the executable images of the system. Use the most recent version, and follow the instructions for installing TLV on your machine.

In addition you may refer to the SMV manual, which is available in the course’s web-page. The relevant part of this manual is the description of the SMV input language.

All solutions to this assignment should be submitted as attachments to an e-mail message. The textual part should be prepared as a postscript, PDF, or Word document. All submitted SMV and PF files should be submitted as separate files. You may group all relevant files into a single file, using ZIP or TAR.

1 Programs and Their Corresponding FDS’s

Task 1: Prove the following claim which is stated in the Lecture notes:

Every FDS which is derived from an SPL program is viable.

Recall that an FDS is viable if any finite run of the system can be extended to a computation. In order to prove this claim it is sufficient (and advisable) to construct a “scheduler” which at any point in the computation will choose among the enabled transitions which transition should be executed next. This choice should guarantee that all the justice and compassion requirements are satisfied, if we consistently follow the scheduler’s recommendations.

There are two important properties of FDS’s derived from programs which are essential for the construction of such scheduler:

• For every justice requirement $J \in \mathcal{J}$, if a state $s$ does not satisfy $J$, it has a successor which satisfies $J$.

• For every compassion requirement $\langle p, q \rangle \in \mathcal{C}$, if a state $s$ satisfies $p$, it has a successor which satisfies $q$.

Prove that these two properties hold for every SPL-derived FDS, and that any system satisfying these properties is viable.
Task 2: Two statements $S_1, S_2$ are said to be congruent, denoted $S_1 \approx S_2$ if the replacement of one of them by the other, in any context, yields equivalent programs, i.e. programs which produce the same set of observations (modulo stuttering). For example the statements $S_1 : x := 1; z := 3$ and $S_2 : (x, z) := (1, 3)$ are not congruent. This is because the program $S_1 \parallel [y := 2]$ can produce the observation:

$$\langle x : 0, y : 0, z : 0 \rangle, \langle x : 1, y : 0, z : 0 \rangle, \langle x : 1, y : 2, z : 0 \rangle, \langle x : 1, y : 2, z : 3 \rangle$$

which cannot be generated by $S_2 \parallel [y := 2]$.

On the other hand, for every statement $S$, $S$ is congruent to $[S; \text{skip}]$.

Consider the following pairs of statements and identify which pairs are congruent. For the cases of non-congruence, provide a context which will distinguish between the two statements.

1. A general $S$ and $[\text{skip}; S]$
2. $x := 1$ and $\text{skip}; x := 1$
3. if $x = 0$ then $S_1$ else $S_2$ and $[\text{await } x = 0; S_1]$

   OR

   $[\text{await } x \neq 0; S_2]$
4. $\text{await } x > 0$ and $[\text{local } \text{done} : \text{boolean}]

   done := 0

   while $\neg$done do

   done := ($x > 0$)


2 Mutual Exclusion With Special Instructions

In the following two questions, you are asked to program an algorithm for mutual exclusion for two processes. Instead of using the semaphore instructions request $x$ and release $x$, we introduce alternative special instructions, applied to a shared variable $x$. In your solutions, you can use $x$ as the only variable shared between the two processes.

2.1 Add-and Store Instructions

Program a mutual exclusion algorithm using the special add-and-store instruction

$$y := x := x + C,$$

where $C$ is a (possibly negative) integer constant, and $y$ is a local variable. This instruction adds in one step the constant $C$ to the shared variable $x$ and copies the updated value of $x$ also to the local variable $y$. 

1
The way one can use this special instruction is demonstrated in the following possible fragment of a mutual exclusion algorithm:

\begin{align*}
  x \text{ : integer where } x &= 0 \\
  \begin{array}{l}
    y_1 \text{ : integer} \\
    \text{loop forever do} \\
    \ell_0 : \text{ Non-Critical} \\
    \ell_1 : y_1 := x := x + 1 \\
    \ell_2 : \text{ if } y_1 = 1 \text{ then} \\
    \quad \ell_3 : \text{ Critical} \\
    \quad \vdots \\
  \end{array}
  \parallel
  \begin{array}{l}
    y_2 \text{ : integer} \\
    \text{loop forever do} \\
    \ell_0 : \text{ Non-Critical} \\
    \ell_1 : y_2 := x := x + 1 \\
    \ell_2 : \text{ if } y_2 = 1 \text{ then} \\
    \quad \ell_3 : \text{ Critical} \\
    \quad \vdots \\
  \end{array}
\end{align*}

Note that if the two processes are at locations \( \ell_1 \) and \( m_1 \) and both try to execute the add-and-store instruction at the same time, only one of them will succeed in getting \( y_i = 1 \) while the other (the one executing second in the interleaving order) will obtain \( y_i = 2 \). Thus, this fragment guarantees mutual exclusion. However it does not automatically guarantees accessibility.

To obtain accessibility, you may consider that at least one of the processes will succeed to get to the critical section. It is suggested that this lucky process takes care of its brother which remained locked out in this round. Note that, in addition to applying the add-and-store instructions to variable \( x \), we may also assign to it any constant or expression not involving \( x \) and also test its value inside an if or an await statement.

In your solution, do not assume any compassion requirements.

**Task 3:** Write an SPL program that implements your solution.

**Task 4:** Translate your SPL program into SMV.

**Task 5:** Using the T1V procedures Invariance, check_deadlock, and Temp_Entail, model check that your solution satisfies the properties of mutual exclusion, absence of deadlock and accessibility for both processes.

### 2.2 Atomic Interchange

Next, consider the special instruction

\[ y :=: x, \]

which in one step interchanges the values of shared variable \( x \) and local variable \( y \). As before, you are requested to design an algorithm which achieves mutual exclusion for two processes using this special instruction. In addition to this instruction, the program may also assign constants to \( x \) and test its value.

No compassion requirements may be assumed.
**Task 6:** Write an SPL program that implements your solution using the atomic interchange instruction.

**Task 7:** Translate your SPL program into SMV.

**Task 8:** Using the TLV procedures Invariance, check_deadlock, and Temp_Entail, model check that your solution satisfies the properties of mutual exclusion, absence of deadlock and accessibility for both processes.