# Analysis of Reactive Systems
# Fall 2002: Assignment No. 2

Due Date: 12.30.02

December 4, 2002

This assignment requires verification of safety and liveness properties, using the TLV and TLPVS systems.

## 1 Using the TLV System

In the course's web-page, you will find the following TLV resources:

1. An executable TLV package for the Solaris operating system for the Sun computers.
2. An executable TLV package for the Linux operating system for PC's.
3. A TLV manual. The relevant sections are: 1 (Introduction), and II.4 (TLV-BASIC).
4. A TLV tutorial. It is suggested to skip Section 2 (Floyd's method), any discussions of simulation (such as subsection 3.2), and the parts which describe how TLV handles LTL (e.g. subsections 4.3 and 5.3).

Each of the executable packages contains the following files:

1. An executable version of TLV, called `tlvsun` and `tlvlinux`, respectively.
2. A file `Rules.tlv` and additional `*.tlv` files, which implement some of the basic facilities your programs may use.

It is recommended that you place these three files in the directory in which you intend to develop and test your algorithms. It is also suggested that the TLV file be renamed `tlv`.

### 1.1 Predefined Dynamic Variables

After an SPL program is compiled and translated, TLV constructs the corresponding FDS representation. Its parameters are stored in the following predefined TLV-BASIC variables:

- The assertion characterizing the *initial condition* is stored in variable `_i`.

- The *transition relation* is stored in variable `total`. Subsection 4.3 of the TLV manual provides more information about the internal structuring of the transition relation. For this assignment, these additional details are not necessary, and you should only use the variable `total` as representing the transition relation.

- The *justice requirements* are stored in the array `_j`[1], ..., `_j`[`_jn`], where `_jn` counts the number of justice requirements.

- The *compassion requirements* are stored in the two arrays `_cp`[1], ..., `_cp`[`_cn`] and `_cq`[1], ..., `_cq`[`_cn`], where `_cp[1.._cn]` stores the $p$-parts, `_cq[1.._cn]` stores the $q$-parts, and `_cn` counts the number of compassion requirements.

Assume that the state variables of the considered system are given by $x_1, \ldots, x_m$. Then, the predefined TLV-BASIC variable `_id` contains the assertion $x'_1 = x_1 \wedge \cdots \wedge x'_m = x_m$. Note that the TLV-BASIC expression `!_id` represents the assertion $x'_1 \neq x_1 \vee \cdots \vee x'_m \neq x_m$.

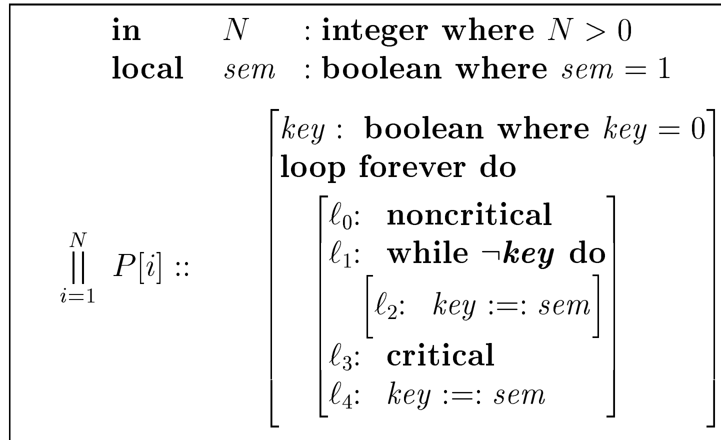
# 2 Mutual Exclusion with Atomic Interchange

Assume an architecture which does not support the semaphore instructions "**request** $y$" and "**release** $y$". Instead, it suports an *atomic interchange* operation of the form

$$x :=: y$$

which, in a single atomic step, interchanges the values of variables $x$ and $y$.

In Fig. 1, we present a program which implements mutual exclusion by using the atomic interchage instruction.



```
in     N   : integer where N > 0
local  sem : boolean where sem = 1

 N
 ||  P[i] ::  [ key : boolean where key = 0
i=1             loop forever do
                  [ l0: noncritical
                    l1: while ¬key do
                          [ l2: key :=: sem ]
                    l3: critical
                    l4: key :=: sem ] ]
```

Figure 1: Program SWAP.

## Task 1: Verify Mutual Exclusion

In the first verification task, you are asked to verify the safety property of mutual exclusion for program SWAP, using TLV. This safety property can be specified by the temporal formula:

$$\Box(i = j \;\vee\; \neg at_\ell_4[i] \;\vee\; \neg at_\ell_4[j])$$

for every $i, j \in [1..N]$.

In Fig. 2, we present the SMV code for program SWAP, taking $N = 6$. It is suggested that

```
MODULE main
DEFINE
  N := 6;
VAR
  sem :  boolean;
  P : array 1..N of process MP(sem);
ASSIGN
  init(sem) := 0;
MODULE MP(sem)
VAR
  loc : 0..4;
  key : boolean;
ASSIGN
  init(loc) := 0;
  init(key) := 1;
  next(loc) :=  case
                  loc = 0          : {0,1};
                  loc in {3,4}     : (loc + 1) mod 5;
                  loc = 1 & key=0  : 3;
                  loc = 1          : 2;
                  1                : 1; -- Here loc = 2
                esac;
  next(key) :=  case
                  loc in {2,4}     : sem;
                  1                : key;
                esac;
  next(sem) := case
                 loc in {2,4}      : key;
                 1                 : sem;
                esac;
JUSTICE
  loc != 1, loc !=2, loc !=3, loc != 4
```

Figure 2: Program `swap6.smv`

you copy this text into a file called "swap6.smv". Alternately, you can copy this file from the web-page.

To perform deductive verification under TLV, create a file "swap-safety.pf" following the template provided in Fig. 3.

```
To compute;
  Print "\n Compute candidate for an inductive assertion \n";
  Let property := 1;
  Let uni1 := 1;
  Let uni2 := 1;
  Let exi  := 0;
  Let i := 1;
  While (i <= N)
    Let uni1 := uni1 & ... ;
    -- Insert in the above \alpha(i)
    Let exi  := exi  | ... ;
    -- Insert in the above \beta(i)
    Let j := 1;
    While (j <= N)
      Let property := property & (i=j | P[i].loc != 3 | P[j].loc != 3);
      Let uni2     := uni2 & ... ;
      -- Insert in the above \gamma(i,j)
      Let j := j+1;
    End -- While (j <= N)
    Let i := i+1;
  End -- While (i <= N)
  Let reach := uni1 & uni2 & exi;
End -- To compute;
compute;
Print "\n Check that computed assertion is inductive \n";
Call binv(reach);
Print "\n Check that 'reach' implies property \n";
Call isvalid(reach -> property);
```

Figure 3: File swap-safety.pf

We assume that the most general inductive asertion for this program will have the form

$$\varphi : \quad \forall i : \alpha(i) \ \wedge \ \exists i : \beta(i) \ \wedge \ \forall i, j : \gamma(i, j)$$

Consequently, file swap-safety.pf contains three blank spaces which you should fill with your choice of $\alpha(i)$, $\beta(i)$, and $\gamma(i, j)$, respectively. In case you think that one of these three forms is unnecessary in order to obtain an inductive assertion, you should fill the respective blank by the constant 1.

## Task 2: Verify Liveness

Next, we would like to prove the liveness property of accessibility. Unfortunately, program SWAP does not guarantee indivisual accessibility. Therefore, we will only verify *communal accessibility* which can be specified by the following response property:

$$(\exists i : at_\ell_1[i]) \quad \Longrightarrow \quad \Diamond\, (\exists j : at_\ell_3[j])$$

This property states that if some process wants to get into the critical section then, eventually, some process (not necessarily the same) will get there.

To prove this property, we will be using rule WELL, presented in Fig. 4. Note that this version of the rule uses a centralized (single) ranking function $\delta$ ranging over the integer interval $[0..k]$.

For a well-founded domain $(\mathcal{A}, \succ)$,
For justice requirements $J_1, \ldots, J_m$,
assertions $p, q = h_0, h_1, \ldots, h_m$,
and ranking function $\delta : \Sigma \mapsto \mathcal{A}$

W1. $p \quad \Longrightarrow \quad \bigvee_{j=0}^{m} h_j$

For $i = 1, \ldots, m$

W2. $h_i \wedge \rho \quad \Longrightarrow \quad (h_i' \,\wedge\, \delta = \delta') \,\vee\, q' \,\vee\, \left(\delta \succ \delta' \,\wedge\, \bigvee_{j=1}^{m} h_j'\right)$

W3. $h_i \quad \Longrightarrow \quad \neg J_i$

---

$p \quad \Longrightarrow \quad \Diamond\, q$

Figure 4: Rule WELL

In Fig. 5 and Fig. 6, we present a "pf" file which prepares the necessary constructs, and then invokes rule WELL.

TLV provides two procedures for the implementation of rule WELL. For the case that the well-founded domain is the natural numbers (or a finite interval of the naturals), the invocation of the rule should be of the form:

```
Call well(start,goal,hset,delta);
```

and the ranking function should be defined by assignments to `delta`. For the more complicated cases, the well-founded domain will be a lexicographic tuple of size $k$, where each component be an interval of the naturals. In such cases, the invocation of the rule should be of the form:

```
Call well-lex(start,goal,hset,delta,k);
```

and the definition of `delta` within the "`pf`" file should assign expressions to the components `delta[1]`,...,`delta[k]`.

As hints to the solution, this file contains the definitions of assertions `in1`, `in2`, `in3`, and `in4`, which hold in a state if one of the processes is currently executing in the respective locations $(1, \ldots, 4)$. The helpful assertions may refer to some of these defined assertions. Also included is a computation of the invariant "`reach`" which may include the inductive assertion computed for the proof of the safety property of this program. It may contain additional assertions which may havce proved unnecessary for establishing the safety property, but may be required for the liveness proof.

### Task 3: Verification diagram

Draw a verification diagram for your liveness proof of program SWAP.

## 3 Ensuring Individual Accessibility

Program SWAP can be extended to a program which does guarantee individual accessibility. This extension is provided by program LIVE-SWAP, presented in Fig. 7.

The program identifies a region, called the *waiting room* which consists of locations $\ell_{1,2}$, and the region called the *competition area* consiting of locations $\ell_{3..7}$. The boolean variable $w[i]$ equals 1 if process $P[i]$ is within the waiting room. Similary, $c[i] = 1$ if $P[i]$ is well into the competition area. The transition from the waiting room into the competition area is controlled by the global variable *gate_open*. The gate is closed by every process leaving the critical section. Then, a process which realizes it is the last to leave the competition (tested at $\ell_9$), opens the gate (at location $\ell_{10}$) and waits there until the waiting room is empty.

Assume that process $P[j]$ enters location $\ell_1$. It may find the gate closed and have to wait at location $\ell_2$. Eventually all processes will leave the competition area, and the last to leave will open the gate, and leave it open at least until $P[j]$ enters location $\ell_5$. Other processes may enter the competition area together with $P[j]$. One of them will win the competition and enter the critical section. When the winning process exits $\ell_7$, it will close the gate which prevents any new competitors from entering the competition. The gate will remain closed until all currently competing processes, including $P[j]$, will visit their critical section.

The `smv` file for program LIVE-SWAP is presented in Fig. 8 and Fig. 9 and .

### Task 4: Safety for Program LIVE-SWAP

Prove the safety property of mutual exclusion for program LIVE-SWAP, specifiable by the formula:

$$\Box\, \forall i \neq j : (\neg at_\ell_7[i] \;\vee\; \neg at_\ell_7[j])$$

### Task 5: Individual Accessibility for Program LIVE-SWAP

Prove the liveness property of individual accessibility for program LIVE-SWAP, specifiable by the formula:

$$at_\ell_1[1] \quad \Longrightarrow \quad \Diamond\, at_\ell_7[1]$$

## Task 6: Verification diagram for Program LIVE-SWAP

Draw a verification diagram for your liveness proof of program LIVE-SWAP.

```
Func trans(line,proc); -- Given a line number and a process number,
-- This function computes the index of the corresponding justice
-- requirement.
  Return line + (proc - 1) * NUMBER_OF_LOCATIONS;
End -- trans(line,proc);
To initialize_justice;
      Print "Initializing the justice arrays\n";
   -- Initializing the justice conditions
   Let j   := JNUM;
   Let in1 := 0;
   Let in2 := 0;
   Let in3 := 0;
   Let in4 := 0;
   While (j)
     Let hset[j]  := 0;
     Let delta[j] := 1;
     Let j := j - 1;
   End -- While (j)
   Let i := 1;
   While (i <= N)
     Let in1      := in1 | P[i].loc=1;
     Let in2      := in2 | P[i].loc=2;
     Let in3      := in3 | P[i].loc=3;
     Let in4      := in4 | P[i].loc=4;
     Let i := i+1;
   End -- While (i <= N)
   Let pend := ... ;
End -- initialize_justice;
To prepare;
  Let NUMBER_OF_PROCESSES := N;
  Let NUMBER_OF_LOCATIONS := 4;
  Let JNUM  := NUMBER_OF_PROCESSES * NUMBER_OF_LOCATIONS;
  initialize_justice;
  Let start := 0;
  Let goal  := 0;
  Let delta := case
                 ...       : 4
                 ...
                 1         : 0;
               esac;
```

Figure 5: File `live6.pf`, Part I

```
  Let i := 1;
  While (i <= N)
    Let start := start | P[i].loc=1;
    Let goal  := goal  | P[i].loc=3;
  -- Requirement (1,i)
    Let j        := trans(1,i);
    Let hset[j]  := ... ;
  -- Requirement (2,i)
    Let j        := trans(2,i);
    Let hset[j]  := ... ;
  -- Requirement (3,i)
    Let j        := trans(3,i);
    Let hset[j]  := ... ;
  -- Requirement (4,i)
    Let j        := trans(4,i);
    Let hset[j]  := ...;
    Let i := i+1;
  End -- While (i <= N)
  Let uni1 := 1;
  Let uni2 := 1;
  Let exi  := 0;
  Let i := 1;
  While (i <= N)
    Let uni1 := uni1 & ... ;
    -- Insert in the above \alpha(i)
    Let exi  := exi  | ... ;
    -- Insert in the above \beta(i)
    Let j := 1;
    While (j <= N)
      Let uni2     := uni2 & ... ;
      -- Insert in the above \gamma(i,j)
      Let j := j+1;
    End -- While (j <= N)
    Let i := i+1;
  End -- While (i <= N)
  Let reach := uni1 & uni2 & (sem -> exi);
End -- To prepare;
prepare;
Call well(start,goal,hset,delta);
```

Figure 6: File live6.pf, Part II

```
in     N           : integer where N > 0
local  sem         : boolean where sem = 1
       gate_open   : boolean where gate_open = 1
       w           : array[1..N] of boolean where w = 0
       c           : array[1..N] of boolean where c = 0

 N
 ||  P[i] ::  [ key : boolean where key = 0
i=1             loop forever do
                  [ ℓ0:  noncritical
                    ℓ1:  w := 1
                    ℓ2:  await gate_open
                    ℓ3:  c := 1
                    ℓ4:  w := 0
                    ℓ5:  while ¬key do
                           [ ℓ6:  key :=: sem ]
                    ℓ7:  critical; c := 0
                    ℓ8:  gate_open := 0
                    ℓ9:  if ∀i : ¬c[i]
                           [ ℓ10: gate_open := 1
                             ℓ11: await ∀i : ¬w[i] ]
                    ℓ12: key :=: sem                ] ]
```

Figure 7: Program LIVE-SWAP.

```
MODULE main
DEFINE
  N  := 6;
  WW := P[1].ww | P[2].ww | P[3].ww | P[4].ww | P[5].ww | P[6].ww;
  CC := P[1].cc | P[2].cc | P[3].cc | P[4].cc | P[5].cc | P[6].cc;
VAR
  sem : boolean;
  gate: boolean;
  P : array 1..N of process MP(sem,gate,WW,CC);
ASSIGN
  init(sem)  := 1;
  init(gate) := 1;
MODULE MP(sem,gate,WW,CC)
DEFINE
VAR
  loc : 0..12;
  key : boolean;
  ww  : boolean;
  cc  : boolean;
ASSIGN
  init(loc)  := 0;
  init(key)  := 0;
  init(ww)   := 0;
  init(cc)   := 0;
  next(loc) :=  case
                  loc=0                 : {0,1};
                  loc in {1,3,4,7,8,10} : loc + 1;
                  loc = 2 & gate  : 3;
                  loc = 5 & !key        : 6;
                  loc = 5               : 7;
                  loc = 6               : 5;
                  loc = 9 & CC          : 12;
                  loc = 9               : 10;
                  loc = 11 & !WW        : 12;
                  loc = 12              : 0;
                  1       : loc;
                esac;
```

Figure 8: File lswap.smv, part I

```
  next(key) :=  case
                  loc in {6,12} : sem;
                  1             : key;
                esac;
  next(sem) := case
                 loc in {6,12}  : key;
                 1              : sem;
                esac;
  next(ww)  := case
                 loc = 1 : 1;
                 loc = 4 : 0;
                 1       : ww;
               esac;
  next(cc)  := case
                 loc = 3 : 1;
                 loc = 7 : 0;
                 1       : cc;
               esac;
  next(gate) := case
                  loc = 8  : 0;
                  loc = 10 : 1;
                  1        : gate;
                esac;
JUSTICE
  loc != 1, loc !=3, loc != 4, loc != 5, loc != 6,
  loc != 7, loc != 8, loc != 9, loc != 10, loc != 12,
  !(loc = 2 & gate), !(loc = 11 & !WW)
```

Figure 9: File `lswap.smv`, part II