Formal Methods of Software Engineering
Fall 2001: Assignment No. 2

Due Date: 12.20.01
December 8, 2001

This assignment requires programming in the TLV-BASIC scripting language, where students are asked to program basic model checking algorithms, using the BDD functionalities provided by the TLV system.

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), II.3 (the SPL input language), II.4 (TLV-BASIC) and the appendices in part III, dealing with the syntax and semantics of SPL, and the error-messages of the SPLC compiler.

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). These parts have been disabled from your version of TLV, because the algorithms you write are supposed to replace some of their functionalities.

Each of the executable packages contains the following files:

1. An executable version of TLV, called tlv\texttt{sun} and tlv\texttt{linux}, respectively.

2. A file Rules.tlv, which implements some of the basic facilities your programs may use.

3. An executable version of the SPLC compiler.

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 \texttt{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 \( \text{total} \). 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 \( \text{total} \) as representing the transition relation.
- The justice requirements are stored in the array \( \_j[1], \ldots, \_j[\_jn] \), where \( \_jn \) counts the number of justice requirements.
- The compassion requirements are stored in the two arrays \( \_cp[1], \ldots, \_cp[\_cn] \) and \( \_cq[1], \ldots, \_cq[\_cn] \), 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 \( x'_1 = x_1 \land \cdots \land x'_m = x_m \). Note that the TLV-BASIC expression \( ! \_id \) represents the assertion \( x'_1 \neq x_1 \lor \cdots \lor x'_m \neq x_m \).

1.2 Special Operations and Functions

In the class presentation, we introduced several notations for performing symbolic operations on state-formulas and relations. In the current context, we use the name state-formula for an assertion which only refers to unprimed variables, and the name relation for an assertions which may also refer to primed variables.

We list below some of the notations introduced in class, and show how they are to be represented in a TLV-BASIC program:

- Let the TLV-BASIC variable \( f \) represent the state-formula \( F \), then the primed version of \( F \) (i.e. \( F' \)) can be represented by the TLV-BASIC expression \( \text{prime}(f) \). Note, that we can only “prime” a state-formula, but never a relation.
- Let \( F \) be an assertion which only refers to primed variables, and assume that it is represented by the TLV-BASIC variable \( f \). Then, the unprimed version of \( F \), obtained by removing the primes from all variables, can be represented by the TLV-BASIC expression \( \text{unprime}(f) \).
- Let the TLV-BASIC variable \( r \) represent the relation \( R \), then the formula \( \exists V : R \) can be represented by the TLV-BASIC expression \( r \text{ forsome } \_vars \).
- Let the TLV-BASIC variable \( r \) represent the relation \( R \), then the formula \( \exists V' : R \) can be represented by the TLV-BASIC expression \( r \text{ forsome primed} \).
• Let $R$ be a relation and $F$ be a state formula, which are represented by the TLV-BASIC variables $r$ and $f$, respectively. Then the immediate $R$-predecessor of $F$, which we denoted in class by $R \circ F$, can be represented by the TLV-BASIC expression \((r \& \text{prime}(f)) \text{ forsome \ vars}\), or more succinctly as $\text{pred}(r,f)$.

• Let $R, F$ and $r, f$, be as above. Then the immediate $R$-successor of $F$, which we denoted in class by $F \circ R$, can be represented by the TLV-BASIC expression \(\text{unprime}((r \& f) \text{ forsome \ vars})\), or more succinctly as $\text{succ}(r,f)$.

Note that we do not provide built-in functions for the iterated predecessor or successor, denoted $R^* \circ F$ and $F \circ R^*$. These, if needed, have to be programmed explicitly.

2 The Programs to be Written

In the following, we list the model-checking algorithms to be written and the test cases on which they should be tried. All of the algorithms should be written on a separate file called my-rules.tlv. Assume that we have prepared an SPL program in file try1.spl and a proof script in file try1.pf. See pages 37,38 of the class presentations for examples of such files. Then the sequence of commands that has to be issued in order to test the correctness of your solution is as follows:

```
  tlv try1.spl
```

The system will print some statistics, and conclude with the statement “your wish is my command”. At which point you should type

```
  load "my-rules.tlv";
  Load "try1.pf";
```

**Task 1:** Write a TLV-BASIC procedure “ProcInvariance(p)” which checks whether assertion $p$ is an invariant of the currently considered SPL program. Test it on program try1, presented in Fig. 1, asking whether the assertion $\neg(at_l \& at_m)$ is an invariant of program try1.

In case the candidate assertion $p$ is not an invariant, the procedure should print a counterexample. This is a sequence of states representing a finite run which leads from an initial state to a state violating $p$.

**Task 2:** Write a TLV-BASIC procedure “To check deadlock” which checks whether the currently considered SPL program can reach a deadlock state. Consult page 42 of the class presentation to see how we characterize a deadlock state, and how this procedure can be invoked. Test your procedure on program try2 presented in Fig. 2, asking first whether this program satisfies the property of mutual exclusion, and then whether it can reach a deadlock situation.

In case the tested program is not deadlock-free, the procedure should print a counterexample. This is a sequence of states representing a finite run which leads from an initial state to a deadlock state.
local y1 : bool where y1 = F;
y2 : bool where y2 = F;

P1::
[%
  l_0: loop forever do [%
    l_1: noncritical;
    l_2: await !y2;
    l_3: y1 := T;
    l_4: critical;
    l_5: y1 := F;
  ]
%
]

P2::
[
  m_0: loop forever do [%
    m_1: noncritical;
    m_2: await !y1;
    m_3: y2 := T;
    m_4: critical;
    m_5: y2 := F
  ]
]

Figure 1: Program try1

Task 3: Write a TLV-BASIC procedure “Proc Temp_Entailment(p,q)” which checks whether the currently considered SPL program satisfies the response property □(p → ◇ q). Consult pages 88 and 96 of the class presentation to see how such algorithms can be written. Test your procedure on program try3 presented in Fig. 3, asking first whether this program satisfies the property of mutual exclusion, then whether it can reach a deadlock situation, and finally whether it satisfies the response property □(at.1.2 → ◇ at.1.3).

In case the tested program does not satisfy the property □(p → ◇ q), the procedure should print a counter-example. A counter-example for a response property consists of a “prefix” α and a “period” β, such that the ultimately periodic sequence obtained by traversing α once and then indefinitely repeating β is a computation violating the response property.

Task 4: Check the properties of mutual-exclusion, deadlock-freedom, and accessibility for the programs MUXSEM (page 28 of class presentation), ATOMIC-PETERSON (page 54 of
local y1 : bool where y1 = F;
y2 : bool where y2 = F;

P1::
  [ l_0: loop forever do [ l_1: noncritical; l_2: y1 := T; l_3: await !y2; l_4: critical; l_5: y1 := F; ] ]

P2::
  [ m_0: loop forever do [ m_1: noncritical; m_2: y2 := T; m_3: await !y1; m_4: critical; m_5: y2 := F ] ]

Figure 2: Program try2

class presentation), BAD-PETERSON (page 55 of class presentation), and GOOD-PETERSON (page 56 of class presentation).

Task 5: The temporal formula $p \implies (\neg r) \ensuremath{\varW} q$ states that, following each occurrence of $p$, $r$ cannot occur strictly before $q$. Thus, this property describes the precedence property “following $p$, $q$ always precedes $r$”. Consider, for example, program GOOD-PETERSON. The precedence property

$$at_{\ell_4} \land at_{\ell_1} \implies (\neg at_{\ell_5}) \ensuremath{\varW} at_{\ell_5}$$

claims that, if process $P_1$ reaches location $\ell_4$ while $P_2$ is still in the non-critical section, then $P_1$ will precede $P_2$ in getting to the critical section. We say that location $\ell_4$ is a priority location for process $P_1$.

Write a TLV-BASIC procedure “Proc Precedes(p,q,r)” which checks whether the currently considered SPL program satisfies the precedence property $p \implies (\neg r) \ensuremath{\varW} q$. Test your
local turn : [1..2] where turn = 1;

P1::
[  
  l_0: loop forever do [  
    l_1: noncritical;  
    l_2: await (turn = 1);  
    l_3: critical;  
    l_4: turn := 2;  
  ]  
]

||

P2::
[  
  m_0: loop forever do [  
    m_1: noncritical;  
    m_2: await (turn = 2);  
    m_3: critical;  
    m_4: turn := 1  
  ]  
]

Figure 3: Program try3

procedure on program GOOD-PETERSON checking whether it satisfies the property

\[ at_{\ell_4} \land at_{m_1} \implies (\neg at_{m_5}) W at_{\ell_5} \]

In case the tested program does not satisfy the precedence property, the procedure should print a counter-example. This is a finite initialized run run which passes through a \(p\)-state followed by an \(r\)-state strictly preceding the next \(q\)-state.

**Task 6:** Does process \(P_1\) in program GOOD-PETERSON possess additional priority locations other than \(\ell_4\) (and \(\ell_5\) which is never considered a priority location). Does process \(P_1\) in program MUXSEM possess any priority locations?

**Task 7:** The temporal formula \(\Box \Diamond p \rightarrow \Box \Diamond q\) characterizes the reactivity property

If there are infinitely many \(p\)-states, then there must be infinitely many \(q\)-states.

Write a TLV-BASIC procedure “Proc React \((p, q)\)” which checks whether the currently considered SPL program satisfies the reactivity property \(\Box \Diamond p \rightarrow \Box \Diamond q\). In case the tested
program does not satisfy the reactivity property, the procedure should print a counter-example. Similarly to the case of a response property, a counter-example here consists of a prefix $\alpha$ and a period $\beta$, such that the ultimately periodic infinite sequence $\alpha\beta^\omega$ violates the property.

Test your procedure on program try3, checking whether it satisfies the property

$$\square \Diamond \at \_\ell_3 \rightarrow \square \Diamond \at \_m_3.$$ 

This property claims that if process $P_1$ visits its critical section infinitely many times, then so does process $P_2$.

Is the same property satisfied by program ATOMIC-PETERTSON?