We will now demonstrate using the programmable symbolic model checker TLV, developed by Elad Shahar, for model checking finite-state systems.

The TLV tool is based on the CMU symbolic model checker SMV and can be used to model check temporal properties of finite-state systems.

Consider the following program: TRY-1, which attempts to solve the mutual exclusion problem by shared variables:

```plaintext
local y1, y2:
boolean

P1 ::

loop forever do
  Non-Critical
  await y2
  Critical
  y1 := 0

P2 ::

loop forever do
  Non-Critical
  await y1
  Critical
  y2 := 0
```

Variables y1 and y2 signify whether processes P1 and P2 are interested in entering their critical sections.

The TLV tool is based on the CMU symbolic model checker SMV and can be used to model check temporal properties of finite-state systems.

Using TLV for Model Checking

We will illustrate the programmable symbolic model checker TLV and illustrate its use on the following example:

**Example of Model Checking:**

We will introduce the following program: TRY-1, which attempts to solve the mutual exclusion problem by shared variables.

```
local y1, y2:
boolean

P1 ::

loop forever do
  Non-Critical
  await y2
  Critical
  y1 := 0

P2 ::

loop forever do
  Non-Critical
  await y1
  Critical
  y2 := 0
```
only later set their own $y$.

Obviously the problem is that the processes test each other's value first and

Reaching the state $\{ \langle 4 \rangle : y_1 = 0, y_2 = 0 \}$, which violates mutual exclusion.

\[
\langle 1 : y_1 = 0, y_2 = 0 \rangle, \langle 1 : y_1 = 0, y_2 = 1 \rangle, \langle 1 : y_1 = 0, y_2 = 0 \rangle
\]

The counter example is:

```
local y1 : boolean where y1 = 0
local y2 : boolean where y2 = 0

P1 ::
  loop forever do
    if Critical then
      y1 := 0
    else
      y2 := 0
    endif
  end

P2 ::
  loop forever do
    if Critical then
      y2 := 0
    else
      y1 := 0
    endif
  end
```

The counter-example follows:

*** Property is NOT VALID ***

Counter-Example:

State 1: $y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 0$
State 2: $y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 1$
State 3: $y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 2$
State 4: $y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 3$
State 5: $y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 3$
State 6: $y_1 = 0, y_2 = 0, P[1].loc = 2, P[2].loc = 3$
State 7: $y_1 = 0, y_2 = 1, P[1].loc = 3, P[2].loc = 4$
State 8: $y_1 = 1, y_2 = 1, P[1].loc = 4, P[2].loc = 4$

Obviously the problem is that the processes test each other's value first and

The call to procedure Invariance checks whether any reachable state violates the assertion exclusion.

The call to procedure Invariance invokes the process which checks whether

The results of model-checking TRY-1 are

```
result of verifying TRY-1:
```
Counter-example follows:

*** Property is NOT VALID ***

Model checking invariance property

Check for the absence of deadlock.

Later, we obtain the following results:

*** Property is VALID ***

Model checking invariance property

Check for mutual exclusion

Mathematically, we can characterize all deadlock states by the assertion:

A state is said to be a deadlock state if no process can perform any action in a state and all processes have no successor different from itself.

Program Properties: Absence of deadlocks

Reaching the deadlock state:

\[
\{(I : y_1 = 1, y_2 = 1, P[1].loc = 3, P[2].loc = 3),
\{(I : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3),
\{(0 : y_1 = 0, y_2 = 1, P[1].loc = 0, P[2].loc = 3),
\{(0 : y_1 = 0, y_2 = 0, P[1].loc = 2, P[2].loc = 2),
\{(0 : y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 2),
\{(0 : y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 2).
\]

The following program TRV-2 is now correct:

The following program TRV-1 Intercache the order of testing and setting:

Second Attempt: Set first and test later

Let us see whether the program is now correct.
Try a Different Approach

The following program TRY-3 uses a variable turn to indicate which process has the higher priority.

```
local turn : [1 :: 2]
where turn = 1
P1 ::

{ turn = 1 } loop forever do
  1: Non-Critical
  2: await turn = 1
  3: Critical
  4: turn := 2

k

P2 ::

{ turn = 2 } loop forever do
  1: Non-Critical
  2: await turn = 2
  3: Critical
  4: turn := 1

m
```

The counter example is:

```

```

The counter-example follows:

*** Property is NOT VALID
check for the absence of deadlock.
*** Property is NOT VALID
```

We obtain the following results:

```
>> Load "try3.pf";
Check for Mutual Exclusion
*** Property is VALID
Check for the absence of Deadlock.
*** Property is VALID
Check Accessibility for P1
*** Property is NOT VALID
Counter-Example Follows:
Stateno.1: P1.loc=0, P2.loc=0, turn=1,
Stateno.2: P1.loc=1, P2.loc=0, turn=1,
Stateno.3: P1.loc=2, P2.loc=0, turn=1,
Stateno.4: P1.loc=3, P2.loc=0, turn=1,
Stateno.5: P1.loc=4, P2.loc=0, turn=1,
Stateno.6: P1.loc=0, P2.loc=0, turn=2,
Stateno.7: P1.loc=1, P2.loc=0, turn=2,
Stateno.8: P1.loc=2, P2.loc=0, turn=2,
Loop back to state 8
```

```
```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
Finally, a good program for mutual exclusion:

Following is a good thread-safe variable solution to the mutual exclusion problem.

Peterson's for 2 Processes:

\[\begin{align*}
\text{local } y_1, y_2; \\
\text{boolean where } y_1 = y_2 = 0; \\
\text{f_1, f_2; where } s = 1; \\
P_1:: \\
0: \text{loop forever do} \\
1: \text{Non-Critical} \\
2: (y_1, s) := (1, 1) \\
3: \text{await } y_2 = 0 \land s = 1 \\
4: \text{Critical} \\
5: y_1 := 0; \\
P_2:: \\
0: \text{loop forever do} \\
1: \text{Non-Critical} \\
2: (y_2, s) := (1, 2) \\
3: \text{await } y_1 = 0 \land s = 2 \\
4: \text{Critical} \\
5: y_2 := 0;
\end{align*}\]

Variables \(y_1\) and \(y_2\) signify whether processes \(P_1\) and \(P_2\) are interested in entering their critical sections. Variable \(s\) serves as a tie-breaker. It always contains the signature of the last process to enter the waiting location (\(t_3, m_3\)). Model checking this program, we find that it satisfies the three properties of (invariance of mutual exclusion, absence of deadlock, and accessibility).

Consider the following example of an SMV code:

Additional Examples of SMV Code