

```
// This is a comment.
```

```
// The following line would have been initialized had we written  
// in i : [0..5] where i = 5;  
in i : [0..5];
```

```
P1::
```

```
[  
  l_0: while (i>0) do [  
    l_1: i := i - 1;  
  ];  
  l_2:  
]
```

```
MODULE main
```

```
VAR
```

```
  pil: {l_0,l_1,l_2}; # control  
  i: 0..5;
```

```
INIT
```

```
  pil = l_0
```

```
DEFINE
```

```
  at_l_0 := pil = l_0;  
  at_l_1 := pil = l_1;  
  at_l_2 := pil = l_2;  
  control_variables :=  
    pil ;  
  control_variables_support :=  
    support(pil) ;
```

```
TRANS
```

```
# l_0
```

```
(  
  pil = l_0 & ( ((i > 0) & next(pil) = l_1) | (!(i > 0) &  
  next(pil) = l_2) ) & next(i) = i  
)
```

```
# l_1
```

```
(  
  pil = l_1 & next(pil) = l_0 & next(i) = (i - 1)  
)
```

```
JUSTICE
```

```
! at_l_0,  
! at_l_1
```

The output of this command looks something like this:

```
% tlv count.spl
TLV version 2.0
Finding transition of main ... This transition is named _t[1],_d[1]

resources used:
user time: 0.0333333 s, system time: 0.0166667 s
BDD nodes allocated: 156
Bytes allocated: 262208
Loaded rules file.
```

Your wish is my command ...

>>

```
>> simulate 100;
Adding 99 new steps to simulation array
Simulation execution terminated at step 13
New simulation created
```

```
>> show_all;
---- Step 1
pi1 = l_0,          i = 5,
---- Step 2
pi1 = l_1,          i = 5,
---- Step 3
pi1 = l_0,          i = 4,
---- Step 4
pi1 = l_1,          i = 4,
---- Step 5
pi1 = l_0,          i = 3,
---- Step 6
pi1 = l_1,          i = 3,
---- Step 7
pi1 = l_0,          i = 2,
---- Step 8
pi1 = l_1,          i = 2,
---- Step 9
pi1 = l_0,          i = 1,
---- Step 10
pi1 = l_1,          i = 1,
---- Step 11
pi1 = l_0,          i = 0,
---- Step 12
pi1 = l_2,          i = 0,
---- Step 13
Halt
```

```
local   x   : bool;
local   y   : bool;
local   z   : bool;
local   i   : [1..4];
```

```
P1::
```

```
[
  l_0: x := T;
  l_1: y := F;
  l_2:
]
```

```
MODULE main
```

```
VAR
```

```
  pi1: {l_0,l_1,l_2}; # control
  x: boolean;
  y: boolean;
  z: boolean;
  i: 1..4;
```

```
INIT
```

```
  pi1 = l_0
```

```
DEFINE
```

```
  at_l_0 := pi1 = l_0;
  at_l_1 := pi1 = l_1;
  at_l_2 := pi1 = l_2;
  control_variables :=
    pi1 ;
  control_variables_support :=
    support(pi1) ;
```

```
TRANS
```

```
# l_0
```

```
(
  pi1 = l_0 & next(pi1) = l_1 & next(x) = TRUE & next(y) = y &
  next(z) = z & next(i) = i
) |
```

```
# l_1
```

```
(
  pi1 = l_1 & next(pi1) = l_2 & next(y) = FALSE & next(x) = x &
  next(z) = z & next(i) = i
)
```

```
JUSTICE
```

```
! at_l_0,
! at_l_1
```

Let dynamic_var := expression;

For example, we can enter the following:

```
>> Let a := x \vee y;  
>> Let b := z /\ y;
```

Expressions

The following table details all the boolean expressions:

operator	TLV
\wedge	&, \wedge
\vee	, \vee
\neg	!
\rightarrow	->
\leftrightarrow	<->

We can also use some expressions which are not part of propositional calculus. For example:

```
>> Let d := i = 3;
```

- start e — create a new simulation which has only the first step (which is the initial state), constrained according to expression e. This command can be used to start a simulation with the input variables initialized with a particular assignment.
- cont n — extend an existing simulation by n steps.
- step e — try to extend an existing simulation by one step, where in the next step, the expression e holds. If there is no such step then the simulation is not extended.
- trunc n — ^{stop} truncate simulation at step n.
- last — print last step of the simulation.
- show n — show a part of a simulation which includes step n.

```
out y : [0..200] where y = 0 ;
local x : [0..1] where x = 0 ;
```

```
P1::[
  l_0: while x=0 do
    [ l_1: y := y + 1];
  l_2:
]
```

```
||
```

```
P2:: [
  m_0: x := 1;
  m_1:
]
```

```
MODULE main
```

```
VAR
```

```
pi1: {l_0,l_1,l_2}; # control
pi2: {m_0,m_1}; # control
y: 0..200;
x: 0..1;
```

```
INIT
```

```
pi1 = l_0 &
pi2 = m_0 &
(x = 0) &
(y = 0)
```

```
DEFINE
```

```
at_l_0 := pi1 = l_0;
at_l_1 := pi1 = l_1;
at_l_2 := pi1 = l_2;
at_m_0 := pi2 = m_0;
at_m_1 := pi2 = m_1;
control_variables :=
  pi1 &
  pi2 ;
control_variables_support :=
  support(pi1) &
  support(pi2) ;
```

```
TRANS
```

```
# l_0
(
  pi1 = l_0 & ( ((x = 0) & next(pi1) = l_1) | (!(x = 0) &
  next(pi1) = l_2) ) & next(pi2) = pi2 & next(y) = y & next(x) = x
) |
# l_1
(
  pi1 = l_1 & next(pi1) = l_0 & next(y) = (y + 1) & next(pi2) = pi2 &
  next(x) = x
) |
# m_0
(
  pi2 = m_0 & next(pi2) = m_1 & next(x) = 1 & next(pi1) = pi1 &
  next(y) = y
)
```

```
JUSTICE
```

```
! at_l_0,
! at_l_1,
! at_m_0
```

```

>> simulate 100;
Adding 99 new steps to simulation
Simulation execution terminated at step 6
New simulation created
>> show_all;
---- Step 1
pi1 = l_0,          pi2 = m_0,          y = 0,          x = 0,
---- Step 2
pi1 = l_1,          pi2 = m_0,          y = 0,          x = 0,
---- Step 3
pi1 = l_1,          pi2 = m_1,          y = 0,          x = 1,
---- Step 4
pi1 = l_0,          pi2 = m_1,          y = 1,          x = 1,
---- Step 5
pi1 = l_2,          pi2 = m_1,          y = 1,          x = 1,
---- Step 6
Halt

```

```

>> trunc 2;
Simulation truncated at step 2
>> show_all;

```

```

---- Step 1
pi1 = l_0,          pi2 = m_0,          y = 0,          x = 0,
---- Step 2
pi1 = l_1,          pi2 = m_0,          y = 0,          x = 0,

```

>> step ! at_m_1;

```

>> step ! at_m_1;
Step 3 was added to simulation
>> step ! at_m_1;
Step 4 was added to simulation
.
.
.

```

```

>> step ! at_m_1;
Step 21 was added to simulation

```

Operator	Name	TLV representation
$\square p$	Henceforth p	\square
$\diamond p$	Eventually p	$\langle \rangle$
$p \mathcal{U} q$	p Until q	p Until q
$p \mathcal{W} q$	p Waiting-for (Unless) q	p Awaits q
$\bigcirc p$	Next p	$()$

Table 1: Future Temporal Operators

Operator	Name	TLV representation
$\sqsupset p$	So-far p	$[-]$
$\diamondleftarrow p$	Once p	$\langle \rangle$
$p \mathcal{S} q$	p Since q	p Since q
$p \mathcal{B} q$	p Back-to q	p Backto q
$\ominus p$	Previously p	$(-)$
$\odot p$	Before p	(\sim)

Table 2: Past Temporal Operators

```
x : [0..10];
```

```
MODULE main
```

```
VAR
```

```
  x: 0..10;
```

```
INIT
```

```
  TRUE
```

```
DEFINE
```

```
  control_variables :=
```

```
  0;
```

```
  control_variables_support :=
```

```
  0;
```

```
x :[0..10];
```

Figure 17: Program emptyx.spl

We can now build a finite representation of an infinite sequence of states:

```
% tlv emptyx.spl
splc: warning: no code for this program.
TLV version 2.0
Finding transition of main ... This transition is empty

resources used:
user time: 0.05 s, system time: 0.0333333 s
BDD nodes allocated: 27
Bytes allocated: 262208
Loaded rules file.
```

Your wish is my command ...

>>

TLV printed several remarks that the program we loaded has no transitions, but we do not need any transitions for this session.

```
>>appstep x = 1 ;
A step has been appended to the simulation
>> appstep x = 2 ;
A step has been appended to the simulation
>> appstep x = 3 ;
A step has been appended to the simulation
>> appstep x = 4 ;
A step has been appended to the simulation
>> appstep x = 5 ;
A step has been appended to the simulation
>> appstep x = 6 ;
A step has been appended to the simulation
>> appstep x = 7 ;
A step has been appended to the simulation
>> appstep x = 8 ;
A step has been appended to the simulation
>> setloop 6;
Loop set to go back to step 6
>> show_all;
---- Step 1
x = 1,
---- Step 2
x = 2,
---- Step 3
x = 3,
---- Step 4
x = 4,
---- Step 5
x = 5,
---- Step 6
x = 6,
---- Step 7
x = 7,
---- Step 8
x = 8,
Loop back to step 6
```

This simulation represents an infinite sequence of states where x has the following values: 1, 2, 3, 4, 5, 6, 7, 8, 6, 7, 8, 6, 7, 8, ...

```
fsimtl ltl( temporal_formula );
```

For example, we can examine the interpretation of the \bigcirc temporal operator. In the following printout the user requests to annotate the simulation with the evaluation of the temporal formula $\bigcirc(x = 6)$ at each step.

```
>> fsimtl ltl(  $\bigcirc(x = 6)$  );
```

```
---- State no. 1 =
```

```
x = 1,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 2 =
```

```
x = 2,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 3 =
```

```
x = 3,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 4 =
```

```
x = 4,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 5 =
```

```
x = 5,  
 $\bigcirc(x = 6)$ 
```

```
---- State no. 6 =
```

```
x = 6,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 7 =
```

```
x = 7,  
!(  $\bigcirc(x = 6)$  )
```

```
---- State no. 8 =
```

```
x = 8,  
 $\bigcirc(x = 6)$ 
```

Loop back to state 6

It turns out that this formula holds only on states 5 and 8. These are both states whose successor is a state where $x = 6$.

Another example — we can use the `fsimtl` command to compare the temporal operators \mathcal{U} , \mathcal{W} :

```
>> fsimtl ltl( ( 3 <= x & x <= 4 | x >= 6 ) Until (x = 5) );
```

```
---- State no. 1 =
```

```
x = 1,
```

```
!( (3 <= x & x <= 4 | x >= 6) Until (x = 5) )
```

```
---- State no. 2 =
```

```
x = 2,
```

```
!( (3 <= x & x <= 4 | x >= 6) Until (x = 5) )
```

```
---- State no. 3 =
```

```
x = 3,
```

```
(3 <= x & x <= 4 | x >= 6) Until (x = 5)
```

```
---- State no. 4 =
```

```
x = 4,
```

```
(3 <= x & x <= 4 | x >= 6) Until (x = 5)
```

```
---- State no. 5 =
```

```
x = 5,
```

```
(3 <= x & x <= 4 | x >= 6) Until (x = 5)
```

```
---- State no. 6 =
```

```
x = 6,
```

```
!( (3 <= x & x <= 4 | x >= 6) Until (x = 5) )
```

```
---- State no. 7 =
```

```
x = 7,
```

```
!( (3 <= x & x <= 4 | x >= 6) Until (x = 5) )
```

```
---- State no. 8 =
```

```
x = 8,
```

```
!( (3 <= x & x <= 4 | x >= 6) Until (x = 5) )
```

Loop back to state 6

```
>> fsimtl ltl( <>(x = 4) & <>(x = 6) );
```

```
---- State no. 1 =
```

```
x = 1,  
<>(x = 4), <>(x = 6)  
<>(x = 4) & <>(x = 6)
```

```
---- State no. 2 =
```

```
x = 2,  
<>(x = 4), <>(x = 6)  
<>(x = 4) & <>(x = 6)
```

```
---- State no. 3 =
```

```
x = 3,  
<>(x = 4), <>(x = 6)  
<>(x = 4) & <>(x = 6)
```

```
---- State no. 4 =
```

```
x = 4,  
<>(x = 4), <>(x = 6)  
<>(x = 4) & <>(x = 6)
```

```
---- State no. 5 =
```

```
x = 5,  
!( <>(x = 4) ), <>(x = 6)  
! ( <>(x = 4) & <>(x = 6))
```

```
---- State no. 6 =
```

```
x = 6,  
!( <>(x = 4) ), <>(x = 6)  
! ( <>(x = 4) & <>(x = 6))
```

```
---- State no. 7 =
```

```
x = 7,  
!( <>(x = 4) ), <>(x = 6)  
! ( <>(x = 4) & <>(x = 6))
```

```
---- State no. 8 =
```

```
x = 8,  
!( <>(x = 4) ), <>(x = 6)  
! ( <>(x = 4) & <>(x = 6))
```

```
Loop back to state 6
```

```
p :bool;
q :bool;
r :bool;
```

Figure 18: Program empty.spl

The command `valid` checks for validity of temporal formulas for both future and tempo operators. The syntax is:

```
valid ltl( temporal_formula );
```

```
>> valid ltl( $\square$ p ->  $\langle$ p);
Model checking...
```

```
*** Property is VALID ***
>>
```

The next example shows a simple case where we check a formula which is not valid:

```
>> valid ltl( $\square$ p);
Model checking...
```

```
*** Property is NOT VALID ***
```

```
Counter-Example Follows:
```

```
---- State no. 1 =
p = 0,
!(  $\square$ p )
```

```
Loop back to state 1
```

```
>> valid ltl(<>p -> []p);  
Model checking...
```

```
*** Property is NOT VALID ***
```

```
Counter-Example Follows:
```

```
---- State no. 1 =  
p = 0,  
<>p, !( []p )
```

```
---- State no. 2 =  
p = 1,  
<>p, !( []p )
```

```
Loop back to state 1
```

```
>> valid ltl( []p -> p );
```

```
>> valid ltl( []p -> ( )p );
```

```
>> valid ltl( []p -> (~)p );
```

```
>> valid ltl( p Until q -> p Awaits q );
```

```
>> valid ltl( p Awaits q -> p Until q );
```

```
>> valid ltl( p Backto q -> p Since q );
```

```
>> valid ltl( ! (p Until r Until q) );
Model checking...
```

```
*** Property is NOT VALID ***
```

```
Counter-Example Follows:
```

```
---- State no. 1 =
```

```
p = 0,          q = 0,          r = 1,
p Until (r Until q), r Until q
```

```
---- State no. 2 =
```

```
p = 0,          q = 1,          r = 0,
p Until (r Until q), r Until q
```

```
Loop back to state 1
```

The counter example is a model which satisfies the original formula $pU r U q$.

On the other hand is the formula $\Box p \wedge \Box \neg p$ satisfiable? :

```
>> valid ltl( ! ( [ ] p & [ ] ! p ) );
Model checking...
```

```
*** Property is VALID ***
```