TLV Tutorial

Elad Shahar

February 16, 2000

Contents

1 Introduction

TLV is a flexible system for program verification. Programs for TLV are written either in SPL[1] or in the input language of SMV[2].

The purpose of this tutorial is to guide the user in using TLV to perform various verification tasks. It is specifically tailored for undergraduate students.

TLV can only handle finite state systems. This makes some of the examples presented in the tutorial seem trivial since some methods of verification make much more sense for infinite state systems. However, presentation of these methods in TLV can allow the users to focus on the principles used rather than being bogged down by the technical issues related to using theorem provers.

1.1 Notation

Throughout this document programs are presented as shown is Fig. 1. When saved in a file, such programs can be read by TLV. All the programs in this tutorial can be found in the system directory of TLV.

```
      x : bool;
in  y,w : [1..4];
out   z : [-10..10] where z =0;

P1::[
      l_0: x := F;
      l_1:
    ]
```

Figure 1: Program notation.spl

User interaction with the program is presented in typewriter font, for example,

```
% tlv notation.spl

      Your wish is my command ...

>> Let i := x;
>> ccc;
The program ccc is not defined
```

The '%' sign at the beginning of the first line is the unix shell prompt. TLV is an interactive program. The prompt of TLV is `'>>'`, therefore, anything which is on a line which starts with `'>>'` has been entered by the user. Lines which do not start with one of these prompts have been printed by TLV.

1.2 Using TLV — common pitfals

- Note that the user must terminate each command with a semicolon. A common mistake when beginning to use TLV is to press the RETURN key, and wait for TLV to respond. However, TLV will not process the command until a semicolon is entered.

- TLV is case sensitive. Enter commands in the same case as they appear in this tutorial. For example, the following command will not work.

  ```
  >> let i := x;
  ```

  The command "Let" must start with a capital letter.

- Be careful when using the arithmetic operator "-". The character "-" can appear in a variable name. If you want to use "-" as an arithmetic operator, then surround it with spaces. For example, the commands

  ```
  >> Let a := 2;
  >> Print a - 1;
  1
  ```

  work as expected. However, if you try the following:

  ```
  >> Let a := 2;
  >> Print a-1;
  Line 5: a-1 undefined
  ```

  TLV prints an error message. TLV looked for a variable called "a-1" instead of treating "-" as an arithmetic operator. Since a variable named "a-1" does not exist, an error message is printed.

1.3 The SPL Input Language

For a high level description for reactive programs, TLV uses the *simple programming language* (SPL).

An SPL program has two main section. The first is the *declaration* section which declares the variables the of the program. The other section is the *body* which contains the statements of the program.

The body can define several processes, but we will start with programs which have only one process. Such programs are equivalent to normal sequential programs.

Declarations

An SPL program starts with a *declaration* which consists of a sequence of *declaration statements* of the form

$$\big[\text{mode}\big]\text{variable}, \ldots, \text{variable} : \text{ type } \big[\textbf{where } \varphi\big]$$

Each declaration statement identifies the mode and type of a list of variables, and optionally, specifies constraints on their initial values.

Specifying the mode of a variable is optional. The mode of each declaration statement may be one of the following

in — specifies variables that are inputs to the program.

local — specifies variables that are local to the program. These variables are used in the execution of the program, but are not recognized outside of it.

out — specifies variables that are outputs of the program.

The program is not allowed to modify (i.e., assign new values to) variables that are declared as **in** variables. The distinction between modes **local** and **out** is mainly intended to help in understanding the program and has no particular formal significance.

The type of the variables can be one of the following:

bool — boolean variable.

[i..j] — An integer variable of range i till j.

Since TLV can only handle finite state programs, there is no unbounded integer type. An integer type is specified by a range of allowed values. Since each program variable can only attain a final number of values, the entire program can attain a final number of states.

The optional assertion φ restricts the initial values of the variables. For *local* and *output* variables it is restricted to be of the form $y = e$, specifying an initial value. For an *in* variable, it may be any constraint on the range of values expected for these variables (this also applies when no mode is specified).

For example, the program from Fig. 1 declares 4 variables. The boolean variable `x`, two input variables `y` and `w` of range 1 till 4, and an output variable `z` of range -10 till 10. The variable `z` has initial value 0.

We refer to variables declared in the program as *program variables*.

Statements

Statements in the body of the program should be labeled. The labels are used as names for the statements in our discussion of the program and later in program specifications and proofs.

Following are some basic statements which are available in SPL:

- $\overline{y} := \overline{e}$ — Assign the list of expressions $\overline{e}$ to the list of variables $\overline{y}$ of the same length and corresponding types.
- **if** c **then** S_1
if c **then** S_1 **else** S_2 — Conditional statements.
- **while** c **do** S
loop forever do — Iteration statements.

1.4 A simple example

In Fig. 2, we present a simple SPL program. The program counts down from the initial value of `i` to 0 and then stops.

```
// 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:
]
```

Figure 2: Program count.spl

Comments in SPL are similar to that of C++. The program contains a comment which starts with `//`. The entire line after the `//` is ignored.

The variable `i` is an `in` variable which ranges over the domain from 0 to 5. Its initial value is unspecified — it can be any value in the domain of the variable. The program starts at location `l_0` and consists of a loop which decreases the value of `i` until it reaches the value 0. The program terminates when it reaches location `l_2`.

This program has only one process named P1.

Loading a program

We can run this program. First we have to load it into tlv. The command `tlv` followed by a file name, runs TLV and loads the program in that file. If the suffix of the file is ".spl" the file should contain a program in SPL.

If the file does not exist in the current directory then TLV searches for it in TLV's system directory. The program count.spl is in TLV's system directory so we could run it by entering the following at the unix shell prompt:

```
% tlv count.spl
```

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 ...

>>
```

TLV loaded the file and printed various statistics such as the time and amount of memory required in order to load the program. Different versions of TLV may print out different statistics and messages — do not be alarmed by this.

TLV does not run the program. It asks the user for further instructions. This is indicated by the prompt `>>` .

To load a new program we must exit TLV and rerun it. The `quit` command exits TLV. So to load a new program from a file named junk.spl we would do the following:

```
>> quit;

% tlv junk.spl
```

Another way to exit TLV is to press `<CTRL-D>` when the input line is empty.

Simulations

After the program has been loaded, we could run simulations of the program by using the simulate command. The syntax of the command is:

$$\boxed{\text{simulate } n;}$$

where n is the number of execution steps which should be carried out. However, it is possible that the program will terminate before the number of requested simulation steps. The execution of a simulation of program count.spl looks as follows:

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

The user requested to create a simulation of 100 steps. The first step is chosen at random such that it will conform to the initial condition, thus `i` can initially have any value from 0 to 5. The simulate command then tries to add an additional 99 steps to the simulation, but since the program terminated, only 13 were generated. We can display the simulation by using the command "show_all".

```
>> 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
```

In the printout of the simulation, each step is displayed as a state which the program is in. In a state, each variable of the program has a specific value. pi1 is a program variable which corresponds to the program location.

In the first step of the simulation (the state which appears under the line `---- Step 1`), the program is at the beginning of the program, at location l_0, and the value of the variable `i` is 5. These values correspond to the initial state of the program. In step 2 the program has moved from location l_0 to location l_1. The execution of the program continues, as the value of `i` decreases gradually until the program halts when it reaches location l_2.

The "Halt" at step 13 indicates that execution cannot continue beyond step 12. Later in the tutorial we will talk about reactive programs which never terminate. When simulations of such programs are printed out they will never contain a "Halt" indication.

You can try to rerun the simulate command and get simulations which start with different initial values for the variable `i`. Running the simulate command erases the previous simulation.

1.5 Expressions and Dynamic Variables

Running programs and checking the results is the way most programmers find bugs in their programs. However, this cannot uncover all bugs. We are interested in performing verification, and for this we need to be able to specify in a formal manner what we want the program to do.

Dynamic Variables

In TLV we mostly use expressions in propositional calculus to assist in the verification of programs. Consider the program in Fig. 3.

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

P1::
[
  l_0: x := T;
  l_1: y := F;
  l_2:
]
```

Figure 3: Program junk.spl

After loading this program with TLV we can express various conditions on the program variables, and store these for future reference. This is done by the command Let.

Let dynamic_var := expression;

For example, we can enter the following:

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

These two lines are assignments to *dynamic variables*. There are two types of variables in TLV. *Program variables* are the variables which are defined by the program. For program JUNK these are `x,y,z` and `i`. Dynamic variables are just containers of propositional expressions over the program variables.

Program variables are usually declared in advance in the SPL program. Dynamic variables spring into existence once they are assigned to. Even arrays of dynamic variables do not need to be declared in advance, so the assignment

```
>> Let a[1] := x /\ y;
```

Puts the propositional expression $x \wedge y$ in the dynamic variable `a[1]` .

Suppose we want to express that `i` has value 2 and the program location is at the start of the program. It is tempting to try the following:

```
>> Let i := 2;
The assigned variable already exists (e.g. as a program variable)

>> Let pi1 := l_0;
The assigned variable already exists (e.g. as a program variable)
```

TLV is not able to perform this assignment. It does not allow assigning to program variables. Only dynamic variables can be assigned to. In contrast to other interpreted environments, the state of TLV is not determined by giving values to the program variables.

The reason for this is that we are not just interested in running the SPL program. We want to be able to talk about the program and express various assertions on its variables. To do this we may need to talk about numerous states of the program. Dynamic variables give us this versatility — each dynamic variable can express a state or even a set of states.

For example, to express the state we wanted we could perform the command:

```
>> Let state1 := i = 2 /\ at_l_0;
```

The expression `at_l_0` is a shortcut in TLV which translates into the expression `pi1 = l_0`.

Expressions

The following table details all the boolean expressions:

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

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

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

But this is just syntactic sugar. All program variables in TLV have finite domain. TLV encodes each non-boolean variable by using several boolean variables. Recall that variable `i` has range from 1 till 4, thus it can be encoded using two internal variables, `i1,i2`, which the user cannot access. Internally, TLV translates the expression `i = 3` to an equivalent expression over `i1` and `i2`.

Other boolean operators on integers are `!=, <, >, >=, <=`. TLV also supports the following arithmetic operators: `+, -, *, /`, and `mod`.

Once a dynamic variable has been defined it can be used in other expressions, for example:

```
>> Let c := d /\ x;
```

c is computed by taking the propositional expression in the dynamic variable d, and then doing a conjunction ($\wedge$) with the program variable x.

Displaying Expressions

We can display the contents of c by using the command form, which has the following syntax:

$\boxed{\text{form } exp;}$

where *exp* is a propositional expression or a dynamic variable. For example,

```
>> form c;

x = 1 & i = 3
```

Boolean variables are treated as integers of range 0 to 1, and thus the value of x is printed as `x = 1`. The `form` command prints the expression using the & and | operators which are the same as /\ and \/ respectively.

Propositional expressions are internally represented in a compact manner which allows fast manipulation of expressions. However, the syntax of the original expression is not preserved. Therefore the form command will not always print exactly what you expect, however, it will print an equivalent expression. For example:

```
>> Let k := x <-> y;
>> form k;

(x = 1 -> y = 1) & (x = 0 -> y = 0)
```

In the current version of TLV this command only works well for very simple expressions.

Evaluating Expressions over a State

The most basic operation we can perform on boolean expressions is to find their truth value given an interpretation to the propositions. We can use states of a simulation as such an interpretation. For example, let us return to program junk.spl :

```
% tlv junk.spl

...
      Your wish is my command ...


>> simulate 100;
Adding 99 new steps to simulation
Simulation execution terminated at step 4
```

```
New simulation created

>> show_all;
---- Step 1
pi1 = l_0,          x = 0,              y = 1,              z = 0,
i = 3,
---- Step 2
pi1 = l_1,          x = 1,              y = 1,              z = 0,
i = 3,
---- Step 3
pi1 = l_2,          x = 1,              y = 0,              z = 0,
i = 3,
---- Step 4
Halt
>>
```

We can evaluate expressions over a specific step of the program using the command `eval`:

$$\boxed{\text{eval } \textit{exp,step\_no};}$$

For example, let us find the truth value of the expression `x & y` in steps 1 and 2:

```
>> eval x & y , 1;
0
>> eval x & y , 2;
1
```

The expression is true only in step 2 since in that step both `x` and `y` have the value 1.

We can also use `eval` to evaluate expressions stored in dynamic variables, for example:

```
>> Let a := at_l_2 | ! x;
>> eval a,1;
1
>> eval a,2;
0
>> eval a,3;
1
```

1.6 The finite nature of TLV

All TLV program variables have a finite domain. This makes a great difference on how expressions are evaluated.

Program Fig. 4 just contains two definitions of program variables with range 0 till 10.

Every time we evaluate an expression we can view this expression as if it also includes constraints on `x` and `y`. For example, in TLV, the following are pairs of commands which compute equivalent expressions:

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

Figure 4: Program integers.spl

```
>> Let a := x = 3;
>> Let a := x = 3 & 0 <= y & y <= 10;

>> Let b := x < 3;
>> Let b := 0 <= x & x < 3;
```

We will use expression extensively throughout this tutorial, so you should keep in mind that every expression also includes the constraints on the ranges of program variables.

2 Floyd's Method

Floyd's method [3][4] for verification of computational programs requires the user to find a set of program locations, and attach an assertion to each of them.

This set of locations is called the *cut points* set. The initial and final program locations should be in the cut points. The cut points should cut all loops of the program.

The assertion related to each cut point characterizes the states at this cut point whenever it is reached.

Given a cut point set and corresponding assertions, TLV can verify that the assertions hold for all cut points which are reachable from the initial state.

To invoke the Floyd method we use the chk_inv command. As parameters we provide an array of cuts and an array of expressions which correspond to the cuts. The chk_inv command verifies that the cuts indeed cut all loops and checks whether all expressions indeed hold of their corresponding cuts. The syntax is

chk_inv cut_array, expression_array, n;

Where `cut_array` is the array of cuts, `expression_array` the array of expressions and `n` the length of the arrays. The length of both arrays should be at least `n`, but could be more (the cuts in indexes larger than `n` will not be checked). If the length of one of the arrays is smaller than `n` then the chk_inv will fail.

2.1 Integer division

The following program has inputs `x` and `y`. It calculates the integer quotient of `x/y` into the output variable `q`, and the integer remainder into the output variable `r`. We will verify this program using Floyd's method.

Upon termination of the program the following formula should hold:

$$x = y * q + r \wedge 0 \leq r \wedge r < y$$

At first we try to make it easy on ourselves. We give just a cut for the final location of the program and state that we indeed get the output we expected:

```
>> Let c[1] := at_l_4;
>> Let e[1] := x = y * q + r  /\  0 <= r  /\  r < y;
```

The array `c` is the array of cuts and the array `e` is the array of expressions. We want to show that the expressions hold at their corresponding cuts upon any execution of the program which starts with a state which conforms to the expression of the cut of the initial state.

We are going to use the assignments into `c[1]` and `e[1]` in order to prove that at location `l_4` the expression in `e[1]` holds, i.e. that when the program terminates, the specification holds.

Now we try to invoke the chk_inv command to check that the specification holds.

```
>> chk_inv c,e,1;
Error: Initial condition should have a cut
>>
```

```
in    x  : [0..100];
in    y  : [0..10];
out   q  : [0..100];
out   r  : [-10..100];


P1::
[
  l_0: (r,q) := (x,0);
  l_1: while (r>y) do [
  l_2:   r := r - y;
  l_3:   q := q + 1;
       ];
  l_4:
]
```

Figure 5: Program intdiv.spl

The command chk_inv accepts three parameters, the array of cuts, the array of corresponding expressions, and the number of cuts we want the command to check. Since we only specified one cut, the third parameter is 1.

However, the execution of chk_inv failed. We must specify a cut on the initial location of the program. We try again by adding another cut:

```
>> Let c[1] := at_l_0;
>> Let e[1] := TRUE;

>> Let c[2] := at_l_4;
>> Let e[2] := x = y * q + r  /\  0 <= r  /\  r < y;

>> chk_inv c,e,2;
Error: There exists an uncut loop.
It includes the following location : pi1 = l_2,
```

We specified two cuts, one for the initial location and one for the termination location of the program. However, our cuts do not cut all loops of the program, therefore the command chk_inv refuses to continue.

Next we add a cut in order to cut the loop:

```
>> Let c[1] := at_l_0;
>> Let e[1] := TRUE;

>> Let c[2] := at_l_1;
>> Let e[2] := x = y * q + r;
```

```
>> Let c[3] := at_l_4;
>> Let e[3] := x = y * q + r  /\  0 <= r  /\  r < y;
```

The first cut is for the initial location and the last cut for the termination location. The second cut cuts the loop.

Trying to verify this we get:

```
>> chk_inv c,e,3;
Error: Counter example found:
It is possible that at cut 2 the corresponding invariant holds
yet in adjacent cut 3 the corresponding expression does NOT hold...
For example, the following can hold at cut 3
pi1 = l_4,          x = 0,              y = 0,              r = 0,
```

What this long printout says is that there is a state where at cut 2 the expression in `e[2]`, `x = y * q + r` , holds but when the execution of the program continues and reaches cut 3 the expression in e[3] does not hold. The last line of the printout shows the values of variables in the state which violates the expression of cut 3.

Looking at `e[3]` we can check why the counter example does not hold. The counter example violates the expression `r < y` which specifies that the remainder is smaller than the divisor. The problem is that if y is 0 then the algorithm is meaningless. We could try to limit y such that it will not be 0. We can do this by limiting y at the initial cut, and also carrying on this limitation to all other cuts. We try another attempt as follows:

```
>> Let c[1] := at_l_0;
>> Let e[1] := y > 0;

>> Let c[2] := at_l_1;
>> Let e[2] := x = y * q + r  /\  y > 0;

>> Let c[3] := at_l_4;
>> Let e[3] := x = y * q + r  /\  0 <= r  /\  r < y  /\  y > 0;

>> chk_inv c,e,3;
Error: Counter example found:
It is possible that at cut 2 the corresponding invariant holds
yet in adjacent cut 3 the corresponding expression does NOT hold...
For example, the following can hold at cut 3
pi1 = l_4,          x = 0,              y = 8,              q = 1,
r = -8,
```

Now the problem is with `r`. `r = -8` violates the condition `0 <= r` in e[3]. The problem is that according to e[2], r can indeed be negative. We add a constraint to e[2] and try again.

```
>> Let c[1] := at_l_0;
>> Let e[1] := y > 0;
```

```
>> Let c[2] := at_l_1;
>> Let e[2] := x = y * q + r  /\  y > 0  /\  r >= 0;

>> Let c[3] := at_l_4;
>> Let e[3] := x = y * q + r  /\  0 <= r  /\  r < y  /\  y > 0;

>> chk_inv c,e,3;
Error: Counter example found:
It is possible that at cut 2 the corresponding invariant holds
yet in adjacent cut 3 the corresponding expression does NOT hold...
For example, the following can hold at cut 3
pi1 = l_4,          x = 64,             y = 8,              q = 7,
r = 8,
```

The values of `y` and `r` violate the condition of e[3] that `r < y`. However, this time the problem is not with the specification. There is indeed a bug in the program (so finally our hard work pays off). The loop terminates too soon when `y` divides `x` with no remainder. In Fig. 6 we present a fixed version of the integer division program

```
in    x  : [0..100];
in    y  : [0..10];
out   q  : [0..100];
out   r  : [-10..10];


P1::
[
  l_0: (r,q) := (x,0);
  l_1: while (r>=y) do [
  l_2:   r := r - y;
  l_3:   q := q + 1;
       ];
  l_4:
]
```

Figure 6: Program intdiv-fix.spl

```
>> Let c[1] := at_l_0;
>> Let e[1] := y > 0;

>> Let c[2] := at_l_1;
>> Let e[2] := x = y * q + r  /\  y > 0  /\  r >= 0;
```

```
>> Let c[3] := at_l_4;
>> Let e[3] := x = y * q + r  /\  0 <= r  /\  r < y  /\  y > 0;

>> chk_inv c,e,3;
In all reachable cuts the invariants are true
The reachable cuts are:
1 2 3
```

chk_inv has passed successfully. The printout indicates that in all the cuts which were reachable, the corresponding expressions indeed holds. In principle some program locations may be unreachable. If a cut was placed in such a position it would have little meaning, and indeed TLV does not attempt to check such cuts.

chk_inv proves only partial correctness. The fact that the cut which cuts the final location of the program is reachable only means that there is **some** execution such that it is reachable. For total correctness it is required that **all** executions terminate.

chk_inv checks all verification conditions automatically. This is done simply by checking all the possibilities of assigning values in the range of the variables. It may be the case that if we increase the range of some of the variables, a program which was verified previously may now contain a bug.

Note that since all of the variables are of finite ranges, we have only verified the program for these specific ranges. In principle it is possible that outside these ranges the program will not be correct.

2.2 Multiplication

Another simple example is presented in Fig. 7. The algorithm performs multiplication by using only addition and subtraction.

```
in     x, y : [-10..10];
local count : [-10..10];
out       z : [-100..100];

P1::
[
  l_0: (z,count) := (0,y);
  l_1: while (count > 0) do [
     l_2: (z,count) := (z + x, count - 1);
  ];
  l_3:
]
```

Figure 7: Program mult.spl

Since there is one loop we again need three cuts — two for the initial and final program location, and one to cut the loop. Using the following cut points, the execution of chk_inv succeeds.

```
>> Let c[1] := at_l_0;
>> Let e[1] := x >=0 /\ y >= 0;

>> Let c[2] := at_l_1;
>> Let e[2] := z = x * ( y - count )  /\  x >= 0  /\  y >= 0  /\ count >= 0;

>> Let c[3] := at_l_3;
>> Let e[3] := z = x * y;

>> chk_inv c,e,3;
In all reachable cuts the invariants are true
The reachable cuts are:
1 2 3
```

Therefore the expression in `e[3]` holds at `l_3` — which is what we wanted to verify.

2.3 Square root

The following program computes the approximate square root of `x`. We want to find an integer `y` such that

$$y^2 \leq x < (y+1)^2$$

```
in        x : [-100..100];
local z1,z2 : [0..100];
out       y : [0..100];

P1::
[
  l_0: y := 0;
  l_1: (z1,z2) := (1,1);
  l_2: while (z1 <= x) do [
     l_3: (y,z2) := (y + 1, z2 + 2);
     l_4: z1 := z1 + z2;
  ];
  l_5:
]
```

Figure 8: Program root.spl

Again we use three cuts to verify the program.

```
>> Let c[1] := at_l_0;
>> Let e[1] := x >= 0;
```

```
>> Let c[2] := at_l_2;
>> Let e[2] :=  y*y <= x /\
               z2 = 2*y + 1 /\
               z1 = (y+1)*(y+1);

>> Let c[3] := at_l_5;
>> Let e[3] :=  y*y <= x  /\  x < (y+1)*(y+1);

>> chk_inv c,e,3;
In all reachable cuts the invariants are true
The reachable cuts are:
1 2 3
```

2.4 Exercises

Q1: The program in Fig. 9 is a simple program which multiplies two numbers.

```
 in x,y: [2..10];
 out z : [0..4];

P1::[
     l_0: z := x * y;
     l_1:
    ]
```

Figure 9: Program mult-simp.spl

It has no loops, so in order to verify it we need only two cuts, one for the start of the program, and one for the termination. However, for the expression of the cut for the termination of the program we specify, erroneously, that the program performs addition.

```
>> Let c[1] := at_l_0;
>> Let e[1] := TRUE;

>> Let c[2] := at_l_1;
>> Let e[2] := z = x + y;
```

Run chk_inv, explain why it was fooled and how can this be corrected.

Q2: Another version of multiplication is presented in Fig. 10.
We perform the following verification attempt:

```
in     x,y  : [-10..10];
local count : [-10..10];
out       z : [-100..100];

P1::
[
  l_0: (z,count) := (0,y);
  l_1: while ( z = x * ( y - count) ) do [
     l_2: if count < 10 then
            l_3: (z,count) := (z + x, count - 1);
  ];
  l_4:
]
```

Figure 10: Program mult-forever.spl

```
>> Let c[1] := at_l_0;
>> Let e[1] := TRUE;

>> Let c[2] := at_l_1;
>> Let e[2] := z = x * ( y - count );

>> Let c[3] := at_l_4;
>> Let e[3] := z = x * y;
```

Run chk_inv. Is the program correct? Why?

Q3: The following program plays a game with a user. The game starts with a pile which contains 33 tokens. The user and computer take turns — with the user starting first. In each step the user or computer subtracts from 1 to 3 tokens from the pile. The player who is forced to take the last token loses the game.

```
local pl : [0..33] where pl = 33;
local round: [0..10] where round = 0;
local usersubtract : [1..3];
out   compwin : bool;

P1::[
  l_0: while (pl > 0) do [
    l_1: round := round + 1;

    // *** User move ***

    l_2: // Determine how much to subtract from pile.
    [
      l_3: usersubtract := 1;
      or
      l_4: usersubtract := 2;
      or
      l_5: usersubtract := 3;
    ];

    // Make sure the number the user subtracts is not too big
    l_6: [ l_7: if usersubtract > pl then l_8: usersubtract := pl; ];

    // User removes tokens from pile
    l_9: pl := pl - usersubtract;

    // Check if computer has won.
    l_10: [ l_11: if pl = 0 then l_12: compwin := T; ];

    // *** Computer move ***

    l_13:
    [
      local compsubtract : [1..3];

      // Move only if pile is not empty
      l_14: if pl > 0 then
      l_15:[

        // Determine how much computer subtracts from pile.
```

```
        l_16: compsubtract := (3 - usersubtract) + 1;

        l_17: // Make sure the number the computer subtracts is not too big.
        [ l_18: if compsubtract > pl then l_19: compsubtract := pl; ];

        // Computer removes tokens from pile
        l_20:  pl := pl - compsubtract;

        // Check if user has won.
        l_21: [ l_22: if pl = 0 then l_23: compwin := F; ];
      ];
    ];
  ];
l_24:
]
```

Program pile.spl

In each iteration of the program the user takes a turn and then the computer takes a turn.

To describe the step of a user we use a selection statement which is of the form

```
l_1:[
  l_2: statement 1
  or
  l_3: statement 2
]
l_4:
```

The meaning of this code segment is that the when starting from program location `l_1`, either statement 1 will be executed or statement 2 will be executed, but not both.

Prove with TLV that the computer always wins.

3 Concurrent Programs

Verification yields greater benefits for concurrent programs since they are more difficult to debug and are difficult to analyze informally.

In this section we extend our knowledge of SPL in order to write concurrent programs. We will then write and simulate various algorithms in order to gain some intuition about the complexities of concurrent programs.

3.1 More about SPL

Cooperation

In SPL, concurrency is introduced by the *cooperation* statement `||`. When two or more processes are separated by `||` they are executed in parallel. As an example consider program Fig. 11.

```
  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:
]
```

Figure 11: Program any-y.spl

The first process, P1, consists of a loop which increases the value of `y` until `x = 0`. The other process, P1, assigns the value 1 to `x`.

The execution is interleaved so at each step, one process is chosen and one step of it is executed.

Await

When there are several processes it is sometimes desired that one process wait for the other processes to complete some task. This is the purpose of the *await* command.
The syntax is

await c

The command causes the process to wait until the boolean expression c becomes true, at which point the command terminates.

3.2 Simulation

Concurrent programs can be difficult to understand. Simulations can clarify how such programs work and help to detect bugs. We have performed some simulations in the previous sections using the simulate and show_all commands, but TLV has more commands related to simulations:

- `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` — truncate simulation at step n.
- `last` — print last step of the simulation.
- `show n` — show a part of a simulation which includes step n.

Multiplication

Returning back to Fig. 7 of Section 2 we can perform a simulation starting with specific values in the input variables by enforcing some condition on the initial state. This is done using the command `start`. For example, to compute the result of multiplying 2 by 3, we could do the following:

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

resources used:
user time: 3.3 s, system time: 0.0333333 s
BDD nodes allocated: 3802
Bytes allocated: 262208
Loaded rules file.

      Your wish is my command ...

>> start x = 3 /\ y = 2;
First step of simulation has been created
>> cont 100;
Adding 100 new steps to simulation
Simulation execution terminated at step 8
>> last;
---- Step 7
pi1 = l_3,          x = 3,              y = 2,              count = 0,
z = 6,
```

The `start` command created the first step of the simulation with the constraint `x = 3 /\ y = 2`, thus assigning values to the input variables. The start command creates a simulation of length 1.

The `cont` command adds new states to an already existing simulation. In the case presented above, we extended a simulation which only has one state, the initial state, where `x` and `y` were assigned values.

The `cont` command only managed to create a small amount of steps before terminating. The last step of the simulation was displayed using the command `last`, and indeed in this printout we get `z = 6`.

Any-y

Is it possible that, for program any-y, the variable `y` can obtain the value 10? We can try to check this using simulations. You can try to perform several simulations, print them, and check the value of y obtained at the end of the program.

However, most of the time the program will stop much before `y` even gets close to 10. This happens since when the simulation is created, at each stage there is a chance that process P2 will be executed.

But we can provide some assistance in guiding the simulation. For example, suppose that after some simulation attempts we get the following output from TLV:

```
>> 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
```

Notice that in the printout of the simulation there are two program variables, pi1 and pi2, which hold the program location for each of the two processes.

We want to change the simulation such that process P2 will not execute. This can be done by truncating the simulation so that steps 3 and on will be removed. This can be achieved by using the `trunc` command:

```
>> 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,
```

Now the simulation is only two steps long. We could use the `cont` command to extend this simulation, however, this could lead very quickly to another execution of process P2, which will cause process P1 to terminate prematurely. Instead we will extend the simulation in a slow and controlled manner by using the `step` command.

The command

```
>> step ! at_m_1;
```

extends the simulation by one step such that in the next step process P2 will not be at location m_1. Thus we can repeat this command many times forcing process P1 to continue until `y` obtains the value 10. Note that pressing the up arrow in TLV restores the previously entered commands to the command line, thus you can easily reenter previously entered commands.

```
>> 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
```

In step 21 the variable `y` indeed has value 10.

This is a simple example of how just blindly trying to produce many simulations is not always enough if we want to check if a program can reach a specific state. This state might be an error state — if the program can reach it then there is a bug in the program. To uncover such bugs we will need stronger techniques, such as model checking, which we will use later in this tutorial.

Finding errors

The program in Fig. 12 changes the values of `x` and `y` in an unpredictable manner. Is it possible to reach a state where $x = 1$ and $y = 1$?

Trying to manually create a simulation that reaches this state would be rather tedious. Instead we can just try to create very long simulations and check them. For example:

```
>> simulate 1000;
Adding 999 new steps to simulation
New simulation created
>> find x=1 & y=1;
The expression was not found in the simulation
```

```
  out y : [0..100] where y = 0 ;
local x : [0..100] where x = 0 ;

P1::[
  l_0: loop forever do [

    [
     l_1: x := x + 1;
     or
     l_2: x := x + 2;
    ];

    l_3: if x > 90 then l_4: x := x - 90;
   ];

  l_5:
]

||

P2:: [
  m_0: loop forever do [

    [
     m_1: y := y + 6;
     or
     m_2: y := y + 9;
    ];

    m_3: if y > 90 then m_4: y := y - 90;
   ];

  m_5:
]
```

Figure 12: Program wander.spl

The `find` command searches the simulation for a step for which the expression in the parameter holds. However, even if we perform many simulations and do not find the state we are looking for, it does not necessarily mean that this state is not reachable. We cannot *prove* a state to be unreachable using simulations alone.

3.3 Mutual exclusion

Concurrent processes need to coordinate their activities. One common case is known as the *mutual exclusion problem*. In this problem there is one resource which must be use by one process at a time, yet this resource is accessible by several processes.

For example, one process may be sending data to a printer. If another process starts to send data as well then the printout may be corrupted. We would like only one process to be able to send data to the printer at any given time.

SPL has special commands for describing an activity which either should or should not be coordinated. These commands do not perform any actual operation, but we will use them to write abstract solutions to the mutual exclusion problem and thus avoiding the need for complex calls to send data to a printer for example.

The special commands are:

- **noncritical** — represents uncoordinated activity. We refer to this as the *noncritical section*.
- **critical** — represents activity which requires coordination. We refer to this as the *critical section*.

A good solution to the mutual exclusion problem should satisfy the following two requirements:

- *Exclusion* — No computation of the program should include a state in which both processes are executing in their critical sections at the same time.
- *Accessibility* — Every state in a computation in which one process is at the exit of the noncritical section, must be followed by another state in which the process is in its critical section.

Fig. 13 is our first attempt to write a program for which mutual exclusion holds.

The problem with this program is that the processes are "taking turns" in entering the critical section. But what if a process wants to enter the critical section yet his turn has not arrived? We can simulate this event by keeping one process in its noncritical section, and forcing the other process to approach the critical section not in its turn. This is shown in the following simulation:

```
>> simulate 1;
Adding 0 new steps to simulation
New simulation created
>> step at_l_1;
Step 2 was added to simulation
>> step at_m_1;
Step 3 was added to simulation
>> step at_m_2;
Step 4 was added to simulation
>> step at_m_3;
The expression does not hold on any successor
>> show_all;
---- Step 1
pi1 = l_0,          pi2 = m_0,          t = 1,
---- Step 2
```

```
local t : [1..2] where t = 1;

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

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: await t = 2;
        m_3: critical;
        m_4: t := 1
    ]
]
```

Figure 13: Program try-mux1.spl

```
pi1 = l_1,          pi2 = m_0,          t = 1,
---- Step 3
pi1 = l_1,          pi2 = m_1,          t = 1,
---- Step 4
pi1 = l_1,          pi2 = m_2,          t = 1,
```

The command `simulate 1;` just added the initial condition to the simulation. From then on we lead the simulation by pointing out the next program location we want the simulation to reach.

At first we cause P1 to enter the non critical section. The process P1 can stay there for ever — this can happen if P1 never requires access to the critical section.

From then on we advance process P2 towards the critical section. However, we do not manage to get the process P2 to enter the critical section. The only way P2 can enter the critical section is that P1 exits the non critical section, but this does not necessarily have to happen. Therefore it is possible that P2 never enters the critical section.

We must avoid this dependency between the two processes.

Another attempt to solve the mutual exclusion problem is presented in Fig. 14.

The variables `y1` and `y2` are used to protect the critical sections. Before trying to enter the

```
local y1 : [0..1] where y1 = 0;
local y2 : [0..1] where y2 = 0;

P1::
[
    l_0: loop forever do [
        l_1: noncritical;
        l_2: await y2 = 0;
        l_3: y1 := 1;
        l_4: critical;
        l_5: y1 := 0
    ]
]

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: await y1 = 0;
        m_3: y2 := 1;
        m_4: critical;
        m_5: y2 := 0
    ]
]
```

Figure 14: Program try-mux2.spl

critical section, each process asks if the other process is already in the critical section — this should be indicated by the value of the corresponding y.

In this case we try to check whether the requirement of mutual exclusion holds:

```
>> simulate 1;
Adding 0 new steps to simulation
New simulation created
>> step at_l_1;
Step 2 was added to simulation
>> step at_m_1;
Step 3 was added to simulation
>> step at_l_2;
Step 4 was added to simulation
>> step at_m_2;
```

```
Step 5 was added to simulation
>> step at_l_3;
Step 6 was added to simulation
>> step at_m_3;
Step 7 was added to simulation
>> step at_l_4;
Step 8 was added to simulation
>> step at_m_4;
Step 9 was added to simulation
>> show_all;
---- Step 1
pi1 = l_0,          pi2 = m_0,          y1 = 0,             y2 = 0,
---- Step 2
pi1 = l_1,          pi2 = m_0,          y1 = 0,             y2 = 0,
---- Step 3
pi1 = l_1,          pi2 = m_1,          y1 = 0,             y2 = 0,
---- Step 4
pi1 = l_2,          pi2 = m_1,          y1 = 0,             y2 = 0,
---- Step 5
pi1 = l_2,          pi2 = m_2,          y1 = 0,             y2 = 0,
---- Step 6
pi1 = l_3,          pi2 = m_2,          y1 = 0,             y2 = 0,
---- Step 7
pi1 = l_3,          pi2 = m_3,          y1 = 0,             y2 = 0,
---- Step 8
pi1 = l_4,          pi2 = m_3,          y1 = 1,             y2 = 0,
---- Step 9
pi1 = l_4,          pi2 = m_4,          y1 = 1,             y2 = 1,
```

We have managed to reach a state where both processes are in their critical section. The problem with this solution for the mutual exclusion problem is that there is a window in which the two processes attempt to enter the critical section together. In this window the protection the variables `y1, y2` offer is inadequate.

To increase the protection we can try to assign values to `y1` and `y2` sooner. This idea is presented in Fig. 15.

However, in this program there is a different problem. A deadlock can occur causing *both* processes to wait for the other to exit from the critical section when in fact none of them are. For example:

```
>> simulate 1;
Adding 0 new steps to simulation
New simulation created
>> step at_l_1;
Step 2 was added to simulation
>> step at_l_2;
Step 3 was added to simulation
```

```
local y1 : [0..1] where y1 = 0;
local y2 : [0..1] where y2 = 0;

P1::
[
    l_0: loop forever do [
        l_1: noncritical;
        l_2: y1 := 1;
        l_3: await y2 = 0;
        l_4: critical;
        l_5: y1 := 0
    ]
]

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: y2 := 1;
        m_3: await y1 = 0;
        m_4: critical;
        m_5: y2 := 0
    ]
]
```

Figure 15: Program try-mux3.spl

```
>> step at_l_3;
Step 4 was added to simulation
>> step at_m_1;
Step 5 was added to simulation
>> step at_m_2;
Step 6 was added to simulation
>> step at_m_3;
Step 7 was added to simulation
>> step at_l_4;
The expression does not hold on any successor
>> step at_m_4;
The expression does not hold on any successor
>> step TRUE;
The expression does not hold on any successor
```

At the end of this simulation neither process can advance. The execution of the command `step TRUE;` demonstrates that the final step of the simulation has no successor.

The problem again happens when both processes try to enter their critical sections together. After executing `l_2` and `m_2` both `y` variables have value 1 and thus both processes get stuck waiting for one of the `y` variables to change to 0. Because of the `await` statements the processes cannot make any assessment of the situation to try to overcome this problem.

The program in Fig. 16 replaces the `await` statements by a loop which tries to break the deadlock.

```
local y1 : [0..1] where y1 = 0;
local y2 : [0..1] where y2 = 0;

P1::
[
    l_0: loop forever do [
        l_1: noncritical;
        l_2: y1 := 1;
        l_3: while y2 = 1 do [
               l_4: y1 := 0;
               l_5: y1 := 1;
             ];
        l_6: critical;
        l_7: y1 := 0;
    ]
]

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: y2 := 1;
        m_3: while y1 = 1 do [
               m_4: y2 := 0;
               m_5: y2 := 1;
             ];
        m_6: critical;
        m_7: y2 := 0
    ]
]
```

Figure 16: Program try-mux4.spl

However, there is a different problem here. It is possible to infinitely execute the loops without entering the critical sections.

For example:

```
>> simulate 1;
Adding 0 new steps to simulation
New simulation created
>> step at_l_1;
Step 2 was added to simulation
>> step at_m_1;
Step 3 was added to simulation
>> step at_l_2;
Step 4 was added to simulation
>> step at_m_2;
Step 5 was added to simulation
>> step at_l_3;
Step 6 was added to simulation
>> step at_m_3;
Step 7 was added to simulation
>> step at_l_4;
Step 8 was added to simulation
>> step at_l_5;
Step 9 was added to simulation
>> step at_l_3;
Step 10 was added to simulation
>> step at_m_4;
Step 11 was added to simulation
>> step at_m_5;
Step 12 was added to simulation
>> step at_m_3;
Step 13 was added to simulation
>> show_all;
---- Step 1
pi1 = l_0,          pi2 = m_0,          y1 = 0,             y2 = 0,
---- Step 2
pi1 = l_1,          pi2 = m_0,          y1 = 0,             y2 = 0,
---- Step 3
pi1 = l_1,          pi2 = m_1,          y1 = 0,             y2 = 0,
---- Step 4
pi1 = l_2,          pi2 = m_1,          y1 = 0,             y2 = 0,
---- Step 5
pi1 = l_2,          pi2 = m_2,          y1 = 0,             y2 = 0,
---- Step 6
pi1 = l_3,          pi2 = m_2,          y1 = 1,             y2 = 0,
---- Step 7
pi1 = l_3,          pi2 = m_3,          y1 = 1,             y2 = 1,
---- Step 8
```

```
pi1 = l_4,          pi2 = m_3,          y1 = 1,             y2 = 1,
---- Step 9
pi1 = l_5,          pi2 = m_3,          y1 = 0,             y2 = 1,
---- Step 10
pi1 = l_3,          pi2 = m_3,          y1 = 1,             y2 = 1,
---- Step 11
pi1 = l_3,          pi2 = m_4,          y1 = 1,             y2 = 1,
---- Step 12
pi1 = l_3,          pi2 = m_5,          y1 = 1,             y2 = 0,
---- Step 13
pi1 = l_3,          pi2 = m_3,          y1 = 1,             y2 = 1,
```

Step 13 is identical to step 7. We continue to execute the steps between 7 and 13 for ever, without entering the critical sections.

The following program, `mux-dek.spl`, is an adequate solution to the problem of mutual exclusion. However, we cannot prove that it is correct using simulations. Simulations only allow us to point to errors of our programs. We cannot use them to prove that the requirement of mutual exclusion hold.

```
local y1 : bool where y1 = F;
local y2 : bool where y2 = F;
       t : [1..2] where t = 1;

P1::
[
    l_0: loop forever do [
        l_1: noncritical;
        l_2: y1 := T;
        l_3: while y2 do [
           l_4: if t = 2 then [
                   l_5: y1 := F;
                   l_6: await t=1;
                   l_7: y1 := T;
                ];
             ];
        l_8: critical;
        l_9: t := 2;
        l_10: y1 := F;
    ]
]

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: y2 := T;
        m_3: while y1 do [
           m_4: if t = 1 then [
                   m_5: y2 := F;
                   m_6: await t=2;
                   m_7: y2 := T;
                ];
             ];
        m_8: critical;
        m_9: t := 1;
        m_10: y2 := F
    ]
]
```

Program mux-dek.spl

3.4 Exercises

In the following questions use simulations (when possible) to check various properties of the program wander.spl in Fig. 12.

Some of the questions may require the creation of long simulations. To display only a part of the simulation which includes step n use the command `show n`. For example, `show 100` will display several steps near step 100.

Q1: Is it possible to reach a state where `y = 18` and `x = 3`?

Q2: Is it possible to reach a state where `y = 20` and `x = 1`?

Q3: What is the highest value of the expression `x + y` you have found in a state of a simulation for program wander.spl .

4 Temporal Logic

In this section we introduce the language of *Propositional Temporal Logic* for specifying properties of reactive systems.

4.1 Propositional Temporal Logic (PTL)

A PTL formula is constructed out of propositional formulas and temporal operators. There are two classes of temporal operators, *future* and *past*. The future and past operators used in TLV are presented in Table 1 and Table 2, respectively.

Operator	Name	TLV representation
$\Box\, p$	Henceforth p	`[]`
$\Diamond\, p$	Eventually p	`<>`
$p \mathcal{U} q$	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
$\boxminus\, p$	So-far p	`[_]`
$\diamondminus\, p$	Once p	`<_>`
$p \mathcal{S} q$	p Since q	`p Since q`
$p \mathcal{B} q$	p Back-to q	`p Backto q`
$\ominus\, p$	Previously p	`(_)`
$\widetilde{\ominus}\, p$	Before p	`(~)`

Table 2: Past Temporal Operators

For the past operators we use underline, "_", in TLV rather than "-" to avoid conflict with the `<->` boolean connective.

Precedence

Temporal operators have higher binding power than the boolean ones. For example,

- `p Until q \/ x Awaits y` is interpreted as `(p Until q) \/ (x Awaits y)`
- `[] p /\ q` is interpreted as `([]p) /\ q`

Temporal operators with one operand have higher binding power than those with two operands. For example,
`[] p Until q` is interpreted as `([]p) Until q`.

Temporal operators with two operands have right associativity. For example,
`p Awaits q Awaits r` is interpreted as `p Awaits (q Awaits r)`.

All other operators in TLV are left associative.

Semantics

A *model* for a temporal formula φ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state provides an interpretation for the variables occuring in φ.

For a given state s_j and propositional formula p we write $s_j \| \models p$ when p is true when evaluated over s_j. Given a model σ we present an inductive definition for the notion of a temporal formula holding at a position $j \geq 0$ in σ, written as $(\sigma, j) \models p$:

For a propositional assertion,

- $(\sigma, j) \models p \iff s_j \| \models p$

For the boolean connectives,

- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$

For the future operators,

- $(\sigma, j) \models \Box\, p \iff$ for all $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models \Diamond\, p \iff$ for some $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$
  and $(\sigma, i) \models p$ for every i such that $j \leq i < k$
- $(\sigma, j) \models p \mathcal{W} q \iff (\sigma, j) \models p \mathcal{U} q$ or $(\sigma, j) \models \Box\, p$
- $(\sigma, j) \models \bigcirc\, p \iff (\sigma, j+1) \models p$

For the past operators,

- $(\sigma, j) \models \boxminus\, p \iff$ for all $0 \leq k \leq j, (\sigma, k) \models p$
- $(\sigma, j) \models \diamondminus\, p \iff$ for some $0 \leq k \leq j, (\sigma, k) \models p$
- $(\sigma, j) \models p \mathcal{S} q \iff$ for some $0 \leq k \leq j, (\sigma, k) \models q$
  and $(\sigma, i) \models p$ for every i such that $k < i \leq j$
- $(\sigma, j) \models p \mathcal{B} q \iff (\sigma, j) \models p \mathcal{S} q$ or $(\sigma, j) \models \boxminus\, p$
- $(\sigma, j) \models \ominus\, p \iff j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models \circledsim\, p \iff j = 0$ or $(\sigma, j-1) \models p$

4.2 Interpreting Temporal Formulas

Temporal formulas can express properties of infinite sequences of states. Computations of reactive systems are such sequences. We will use Temporal Logic to specify the desired behavior of a system, by specifying sets of acceptable computations.

But as for now we will try to gain more intuition about Temporal Logic using TLV to evaluate temporal formulas over sequences which are not necessarily computations.

To do this we present some more commands which create simulations, however, these commands allow the user to create arbitrary sequences which are unrelated to the program which tlv has been loaded with:

- `setstep n,e` — replace step `n` of the simulation by a new state on which the expression `e` holds.
- `appstep e` — append a new step to the end of the simulation such that the expression `e` holds on the new step.
- `setvar n,v,e` — in step `n`, change the value of variable `v` to be `e`.
- `setloop n` — create a loop in the simulation such that step `n` is a successor of the last step of the simulation. To cancel the loop use the command `setloop 0`.

The commands `setstep` and `appstep` modify or add steps to the simulation. The value of the variables can be whatever the user desires with no consideration of the program currently loaded by TLV.

The `setvar` commands lets the user modify the value of a single variable in a step. For example, recall the program pile.spl from page 21. We can perform a simulation starting with a different number of tokens in the pile, without changing the program:

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

resources used:
user time: 0.133333 s, system time: 0.0333333 s
BDD nodes allocated: 5898
Bytes allocated: 327744
Loaded rules file.

      Your wish is my command ...

>> simulate 1;
Adding 0 new steps to simulation
New simulation created

>> show_all;
---- Step 1
pi1 = l_0,          pl = 33,            round = 0,          usersubtract = 3,
```

```
compwin = 0,        compsubtract = 3,

>> setvar 1,pl,32;
Variable pl of step 1 has been set

>> show_all;
---- Step 1
pi1 = l_0,          pl = 32,            round = 0,          usersubtract = 3,
compwin = 0,        compsubtract = 3,

>> cont 100;
Adding 100 new steps to simulation
Simulation execution terminated at step 92
```

In this session the user created a simulation of one step and then changed the number of tokens in the pile to 32 using the command `setvar 1,pl,32`. To continue the simulation from where it stopped the user issues the `cont` command.

The `setloop` command is needed so we will be able to represent infinite sequences of states using a finite representation.

Building a model

In the following session we only need program variables so we can use them to hold a state. We use the program in Fig. 17 which has only one declared variable, `x`, with a finite number of possible integer values.

```
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, ...`

Future temporal operators

If we limit ourselves to using only future temporal operators we can demonstrate how different temporal formulas are interpreted along a simulation. For this we use the `fsimtl` command which displays the current simulation annotated with temporal formulas. `fsimtl` is an acronym for "Future-limited version of SIMulations annotated by Temporal Logic". The syntax is

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( ()(x = 6) );

---- State no. 1 =
x = 1,
!( ()(x = 6) )

---- State no. 2 =
x = 2,
!( ()(x = 6) )

---- State no. 3 =
x = 3,
!( ()(x = 6) )

---- State no. 4 =
x = 4,
!( ()(x = 6) )

---- State no. 5 =
x = 5,
()(x = 6)

---- State no. 6 =
x = 6,
!( ()(x = 6) )

---- State no. 7 =
x = 7,
!( ()(x = 6) )

---- State no. 8 =
x = 8,
()(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( ( 3 <= x & x <= 4 | x >= 6 ) Awaits (x = 5) );
```

```
---- State no. 1 =
x = 1,
!( (3 <= x & x <= 4 | x >= 6) Awaits (x = 5) )

---- State no. 2 =
x = 2,
!( (3 <= x & x <= 4 | x >= 6) Awaits (x = 5) )

---- State no. 3 =
x = 3,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

---- State no. 4 =
x = 4,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

---- State no. 5 =
x = 5,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

---- State no. 6 =
x = 6,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

---- State no. 7 =
x = 7,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

---- State no. 8 =
x = 8,
(3 <= x & x <= 4 | x >= 6) Awaits (x = 5)

Loop back to state 6
```

The same formula was evaluated twice, but in the second time the `Until` was replaced by `Awaits`. In both cases the formula holds at states 3 till 5 because in these states it is indeed the case that $3 \leq x \leq 4$ until $x = 5$. However, after state 5 the situation is different.

Note that in the loop of steps 6,7,8 the variable `x` never obtains the value 5, thus the `Until` formula does not hold at these states. In contrast the weaker `Awaits` version holds on these states since the assertion $3 \leq x \leq 4 \vee x \geq 6$ holds indefinitely.

For more complex temporal formulas `fsimtl` provides more information. `fsimtl` annotates the simulation not only with the interpretation of the requested formula, but also with each of its subformulas whose principal operator is temporal. For example:

```
>> 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
```

This simulation is annotated with the interpretation of three temporal formulas: $\Diamond(x = 4) \wedge \Diamond(x = 6)$, $\Diamond(x = 4)$, $\Diamond(x = 6)$. The last two forumlas are subformulas of the first formula (which is the formula we are really interested in) and have a $\Diamond$ as their principal operator. This additional information helps to reason why the entire formula was evaluated the way it was.

Similarly, we can observe the way other future temporal operators work. The following are some interesting temporal formulas you can try:

```
>> fsimtl ltl( [](x > 3) );

>> fsimtl ltl( ()[](x > 5) );
```

Past temporal operators

TLV cannot evaluate temporal formulas with past operators on a sequence which contains a loop. Therefore, before we continue we remove the loop:

```
>> setloop 0;
Loop back has been canceled
```

On top of this we use a different command for evaluating temporal formulas — instead of `fsimtl` we use `simtl`. The command behaves the same as `fsimtl`, it even does its best to try to interpret future temporal formulas as well, however, the semantics of future temporal operators for finite sequences[5] is different from what was described above.

For example, formulas of the type $\Box\varphi$ are always false on finite sequences. On the other hand $\Diamond\varphi$ is true for state j of the sequence iff for some state larger or equal to j the formula φ holds. Thus $\Diamond$ for finite sequences is similar to the `find` command for simulations.

The semantics for finite sequences tries to approximate from below the answer for an infinite sequence. If a formula holds for the finite semantics, on a finite sequence f, then it should also hold using the regular semantics on all infinite extensions of f. On the other hand, if a formula does not hold for the finite semantics on finite sequence f, there still may be infinite extensions for f such that the temporal formula *does* hold.

You can use future temporal operators in the `simtl` command, but be aware that you may not get what you expected. Anyway, there are still plenty of purely past temporal formulas to try! The following are some interesting examples:

```
>> simtl ltl( (_)(x <= 3) );

>> simtl ltl( (~)(x <= 3) );

>> simtl ltl( [_](x <= 3) );

>> simtl ltl( <_>(x = 4) );

>> simtl ltl( (x <= 6) Since (x = 3) );

>> simtl ltl( (x <= 6) Backto (x = 3) );
```

4.3 Validity and Satisfiability

A temporal formula φ is *valid* if it holds over *all* models σ. A temporal formula φ is *satisfiable* if it holds over *some* model σ.

In this section we will use TLV to check the validity and satisfiability of temporal formulas. For this we do not require a running program since we are not checking whether the temporal formula holds for a computation of a program. All we need is to declare some program variables to use as propositions. For this we use the program in Fig. 18.

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

Figure 18: Program empty.spl

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

$$\boxed{\text{valid ltl(temporal\_formula);}}$$

If the temporal formula is valid then a message is printed. If it is not valid then a sequence of states (a model) for which the formula is false is printed.

We start by checking a simple example. Is the formula $\Box\, p \rightarrow \Diamond\, p$ valid? We can check it as follows:

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

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

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

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

*** Property is NOT VALID ***

Counter-Example Follows:

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

Loop back to state 1
```

A counterexample is printed out. It is annotated with the evaluation of all the subformulas which have a principle operator which is temporal. Note that the states of the model only consists of the values of the propositional variables (in this case, the variable `p`). The evaluation of the temporal formulas is only printed to help the user understand why complex temporal formulas are not valid.

Previously we checked that the formula $\Box p \rightarrow \Diamond p$ is valid. Is the formula $\Diamond p \rightarrow \Box p$ valid as well? We can check this as follows:

```
>> 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
```

Since in state 2 `p = 1` then $\Diamond p$ always holds, but since in state 1 `p = 0` then $\Box p$ never holds. Therefore the formula $\Diamond p \rightarrow \Box p$ is false for all states in this model. In particular it is false in state 1, and that is why $\Diamond p \rightarrow \Box p$ is not valid.

Note that validity requires that the formula holds on the *first* state of all models. It may or may not hold on the other states of the model — this does not affect its validity.

Several other interesting validity checks you can try are:

```
>> 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( <>(q & ()q)  -> (TRUE Backto q -> TRUE Since q) );
```

TLV does not have a special routine for checking satisfiability, however, a formula is satisfiable iff its negation is not valid. Therefore it is easy to check for satisfiability: just check for the validity of the negation of the formula, and reverse the result.

For example to check whether $p\mathcal{U}r\mathcal{U}q$ is satisfiable we check the validity of its negation:

```
>> 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 $p\mathcal{U}r\mathcal{U}q$.

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

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

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

Note that the validity check succeeded — the `valid` command printed that the property is valid. However, we are interested in satisfiability. Thus we have to reverse the result, therefore the formula $\Box\, p \wedge \boxminus \neg p$ is *not* satisfiable.

4.4 Exercises

Q1:

When one sneezes, it is polite for others to say "bless you". However, when the same person sneezes again, it becomes impolite to bless him since it would make the sneezer uncomfortable for bothering everybody else. Express this in temporal logic using the propositions *sneeze* and *bless*.

Note: blessing is done one step after sneezing. Also, blessing for no apparent reason is perhaps not impolite, but it is strange nonetheless.

Q2: For each of the following temporal formulas find a model for which it holds (if such a model exists).

1. $\Box \Diamond x = 0 \wedge \Box \Diamond x = 1$
2. $x = 1 \wedge ((\Box(x = 1))\,\mathcal{U}\,(x = 0))$
3. $x = 0 \wedge \Box(x = 0 \rightarrow \bigcirc(x = 1)) \wedge \Box(x = 1 \rightarrow \bigcirc(x = 2)) \wedge \Box(x = 2 \rightarrow \bigcirc(x = 0))$

Q3:

Previously we defined infinite sequences of states by providing a sequence which has a tail which repeats indefinitely. Let h and t (which stand for head and tail) be two finite sequences of state valuations: $h = h_1, \ldots, h_m \quad t = t_1, \ldots, t_n$. To define an infinite sequence of states we use the following syntax:

$$h_1, \ldots, h_m\,(t_1, \ldots, t_n)^\omega = h_1, \ldots, h_m, t_1, \ldots, t_n, t_1, \ldots, t_n, t_1, \ldots$$

For each of the following pairs of models find a temporal formula which is interpreted as false on one of the models and as true on the other.

1. $p = 0, (1, 0)^\omega$ and $p = 0, (1, 1)^\omega$
   (do not use $\bigcirc, \ominus, \circleddash$).

2. $(p, q) = \quad (1, 0), (1, 0), (1, 0), (1, 1), \Big((1, 0), (0, 0), (1, 0), (1, 1)\Big)^\omega$
   and
   $(p, q) = \quad (1, 0), (0, 0), (1, 0), (1, 1), \Big((1, 0), (0, 0), (1, 0), (1, 1)\Big)^\omega$
   (do not use $\bigcirc, \ominus, \circleddash$).

3. $p = 0, 0, 1, (0)^\omega$ and $p = 0, 0, 0, 1, (0)^\omega$
   (do not use $\bigcirc$)

5 Model Checking

Given a finite state SPL program P and a temporal logic specification φ we want to decide whether φ is valid over finite state program P, i.e. whether all the computations of P satisfy φ. The problem of deciding whether φ is valid over P is known as *Model Checking*.

In this session we will use TLV to perform model checking in two different ways. First we will build testers manually, compose them with our program and check for feasibility. Later on we will use TLV to build testers automatically.

5.1 Expansion formulas

The requirement that a temporal formula hold at position j of a state sequence can often be decomposed into two other requirements —

- A simpler formula which should hold at the same position.
- Another formula which should hold at either $j+1$ or $j-1$.

Expansion formulas will be useful in the building of testers.
The following is a list of expansion formulas:

$$\begin{array}{lcl}
\square\, p & \Longleftrightarrow & p \wedge \bigcirc \square\, p \\
\diamondsuit\, p & \Longleftrightarrow & p \vee \bigcirc \diamondsuit\, p \\
p\mathcal{U}q & \Longleftrightarrow & q \vee [p \wedge \bigcirc(p\mathcal{U}q)] \\
p\mathcal{W}q & \Longleftrightarrow & q \vee [p \wedge \bigcirc(p\mathcal{W}q)]
\end{array}$$

$$\begin{array}{lcl}
\boxminus\, p & \Longleftrightarrow & p \wedge \widetilde{\ominus}\, \boxminus\, p \\
\diamondminus\, p & \Longleftrightarrow & p \vee \ominus \diamondminus\, p \\
p\mathcal{S}q & \Longleftrightarrow & q \vee [p \wedge \ominus(p\mathcal{S}q)] \\
p\mathcal{B}q & \Longleftrightarrow & q \vee [p \wedge \widetilde{\ominus}\,(p\mathcal{B}q)]
\end{array}$$

5.2 Building Testers Manually

A tester has several components which need to be defined. In TLV this is done as follows:

- The set of variables:

  ```
  Let tester.vars := vset(<list of variables>)
  ```

  The function `vset` takes a list of variables and returns them in a set representation. Note that besides the variables which are explicitly defined in SPL programs the variables containing the program locations are implicitly defined as `pi1, pi2, pi3,...` according to the number of processes which are defined by the SPL program. For example, for a program with two processes only `pi1` and `pi2` are defined in addition to the explicitly defined variables.

- The initial condition:

  ```
  Let tester.i := <initial condition>
  ```

- The transition relation:

  ```
  Let tester.t := <transition relation>
  ```

- The justice set:

```
Let tester.jn := <number of justice requirements>
Let tester.j[number] := <justice requirement>
```

- The compassion set:

```
Let tester.cn := <number of compassion requirements>
Let tester.c[number] := <compassion requirement>
```

Mutual Exclusion

In Fig. 19 we provide another solution to the mutual exclusion problem. The solution uses a *semaphore* (the integer variable `y`) to protect the critical section. Two new SPL commands are used to deal with semaphores.

- **`request y`** — can be executed only when `y` is positive. When executed, it decreases `y` by 1.
- **`release y`** — increase `y` by 1.

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

P1::
[
    l_0: loop forever do [
        l_1: noncritical;
        l_2: request y;
        l_3: critical;
        l_4: release y
    ]
]

||

P2::
[
    m_0: loop forever do [
        m_1: noncritical;
        m_2: request y;
        m_3: critical;
        m_4: release y
    ]
]
```

Figure 19: Program mux-sem.spl

In program mux-sem.spl, the semaphore, y, has value 0 when one of the processes enters the critical section. The request statement blocks the other process from entering the critical section.

We would like to build a tester to verify mutual exclusion for program mux-sem.spl, however, we need some more knowledge in TLV to do this.

First of all, to create a tester we may need to create new variables. This is done very similar to the way variables are defined in the SPL language, however, it is possible to define new variables directly from TLV's command line. For example:

```
>> Q : boolean;
>> X : 1..3;
```

Thus we defined two new variables, the first Q is boolean, and the second X is a finite integer variable ranging from 1 to 3. It is recommended to use uppercase letters for variables defined in this manner so as not to use names of other program variables or dynamic variables.

We also need to be able to refer to a *second copy* of the program variables — we will refer to variables of the second copy as *primed variables*. We use them to define the transition relation of the tester. The primed variables refer to the state of the program after the transition takes place. To refer to such a variable we use the syntax next(y), where y is a program variables. For example, if we want to define a transition which preserves the value of y and nothing else, we can write this as:

```
>> Let tester.t := y = next(y);
```

Finally, we can build a tester for verifying mutual exclusion:

```
% tlv mux-sem.spl
...
      Your wish is my command ...

>> Q : boolean;

>> Let tester.vars := vset(Q,pi1,pi2);

>> Let tester.i := !Q;

>> Let tester.t := Q <-> (  !(at_l_3 /\ at_m_3) /\ next(Q) ) ;

>> Let tester.jn := 1;
>> Let tester.j[1] := Q \/ (at_l_3 /\ at_m_3);

>> Let tester.cn := 0;
```

To compose the tester with the program and perform model we use the command mc_tester as follows:

```
>> mc_tester;
```

```
Model checking...

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

And indeed, mutual exclusion holds. We can modify the tester to verify different properties. For example, instead of verifying mutual exclusion of the program locations `l_3` and `m_3` we can do the same for `l_2` and `m_3` as in the following session:

```
>> Let tester.t := Q <-> (  !(at_l_2 /\ at_m_3) /\ next(Q) ) ;

>> Let tester.j[1] := Q \/ (at_l_2 /\ at_m_3);

>> mc_tester;

Model checking...

*** Property is NOT VALID ***

Counter-Example Follows:

---- State no. 1 =
Q = 0,              pi1 = l_0,          pi2 = m_0,          y = 1,

---- State no. 2 =
Q = 0,              pi1 = l_0,          pi2 = m_1,          y = 1,

---- State no. 3 =
Q = 0,              pi1 = l_0,          pi2 = m_2,          y = 1,

---- State no. 4 =
Q = 0,              pi1 = l_0,          pi2 = m_3,          y = 0,

---- State no. 5 =
Q = 0,              pi1 = l_1,          pi2 = m_3,          y = 0,

---- State no. 6 =
Q = 0,              pi1 = l_2,          pi2 = m_3,          y = 0,

---- State no. 7 =
Q = 0,              pi1 = l_2,          pi2 = m_4,          y = 0,

---- State no. 8 =
Q = 0,              pi1 = l_2,          pi2 = m_0,          y = 1,

---- State no. 9 =
```

```
Q = 0,              pi1 = l_3,          pi2 = m_0,          y = 0,

---- State no. 10 =
Q = 0,              pi1 = l_4,          pi2 = m_0,          y = 0,

Loop back to state 1
```

A counter example is presented. It shows that there is a computation which has state where the program is at locations `l_2` and `m_3`. Although a finite sequence of states would have been enough to show that the property we tried to verify is not valid, TLV always returns a computation, an infinite sequence of states, as a counter example.

Similarly we can build a tester for verifying accessibility:

```
>> G : boolean;

>> Let tester.vars := vset(G,pi1,pi2);

>> Let tester.i := !G;

>> Let tester.t := G -> ( at_m_2 /\ next(G) ) ;

>> Let tester.jn := 1;
>> Let tester.j[1] := G ;

>> Let tester.cn := 0;

>> mc_tester;

Model checking...

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

5.3 Building Testers Automatically

TLV can build testers automatically from temporal logic specifications. The syntax is:

$\boxed{\text{mc ltl(temporal\_formula);}}$

This command should only be used for reactive systems — programs which never terminate. Verifying mutual exclusion can simply be done as follows:

```
>> mc ltl( [] !(at_l_3 & at_m_3));
Model checking...

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

At this stage we have two formal languages: SPL for writing the implementation, and temporal logic for the specification. Both the implementation and the specification may contain errors. In practice verification is an iterative process where both the implementation and the specification are continually refined in order to increase the reliability of the program.

Assume that we are instructed by a coworker to verify that the program in Fig. 20 does not get stuck forever in some part of the program. This is a very informal specification. How can we formalize it?

```
y : bool where y = F;
z : bool where z = T;
x : [0..3] where x = 1 ;

P1::[
  l_0: loop forever do [
    l_1: (y,x) := (!y,0);
    l_2: while (z) do [
      l_3: x := x + 1;
      l_4: if x = 3 then
        l_5: x := 0;
    ];
   ];
  l_6:
]
```

Figure 20: Program nostuck.spl

Intuitively, one way would be to verify that the variables x and y obtain some values infinitely often. For example:

```
>> mc ltl([]<>y & []<>x=2);
Model checking...

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

Is this satisfactory? Observing the inner loop it seems that x does change its values. As for y it started with value false, so the fact that y is true infinitely often seems like a good indication. However, upon observation of the program we see that the variable z is never assigned to; and since its initial value is true then the internal loop will never be exited. We test our suspicions by changing the specification as follows:

```
>> mc ltl([]<>y & []<>!y & []<>x=2);
Model checking...

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

```
Counter-Example Follows:

---- State no. 1 =
pi1 = l_0,          y = 0,              z = 1,              x = 1,

---- State no. 2 =
pi1 = l_1,          y = 0,              z = 1,              x = 1,

---- State no. 3 =
pi1 = l_2,          y = 1,              z = 1,              x = 0,

---- State no. 4 =
pi1 = l_3,          y = 1,              z = 1,              x = 0,

---- State no. 5 =
pi1 = l_4,          y = 1,              z = 1,              x = 1,

---- State no. 6 =
pi1 = l_2,          y = 1,              z = 1,              x = 1,

---- State no. 7 =
pi1 = l_3,          y = 1,              z = 1,              x = 1,

---- State no. 8 =
pi1 = l_4,          y = 1,              z = 1,              x = 2,

---- State no. 9 =
pi1 = l_2,          y = 1,              z = 1,              x = 2,

---- State no. 10 =
pi1 = l_3,          y = 1,              z = 1,              x = 2,

---- State no. 11 =
pi1 = l_4,          y = 1,              z = 1,              x = 3,

---- State no. 12 =
pi1 = l_5,          y = 1,              z = 1,              x = 3,

Loop back to state 3
```

Observing the counter example we see that `y` never changes after state 3! The program is stuck in the internal loop. Something is wrong with the program. The problem is that we were not told what the program should do, however, we can make an educated guess. The variable `z` is not used in the program. Perhaps we should change location `l_5` as in Fig. 21.

But perhaps not. More information is needed. What is the program supposed to do? Is the informal specification too informal? Perhaps what should really be verified is whether your coworker gave you the right instructions...

```
y : bool where y = F;
z : bool where z = T;
x : [0..3] where x = 1 ;

P1::[
  l_0: loop forever do [
    l_1: (y,x) := (!y,0);
    l_2: while (z) do [
      l_3: x := x + 1;
      l_4: if x = 3 then
        l_5: z := F;
    ];
   ];
  l_6:
]
```

Figure 21: Program nostuck2.spl

5.4 Exercises

Q1: Recall program `any-y.spl` from Fig. 11. Is its FKS feasible? Why?

Q2: The programs in Fig. 22 and Fig. 23 count modulo 4 and 5 respectively. Build a tester manually to verify that whenever the program location is at `l_1` and x is 2 then the number of iterations of the program is even. Model check your tester with the two programs to check your tester. Note that a new iteration starts when the program moves from location `l_0` to `l_1`, thus when the program reaches location `l_1` for the first time then we are in the first iteration.

Hint: Define a new variable (for the tester, not the program) that will keep track (by adding an expression to the tester's transition relation) of whether we are in an odd or even iteration. Furthermore, find out what temporal formula you are required to verify and merge it into the tester.

Q3:

Verify total correctness of program `pile.spl` from Fig. 2.4 without creating the tester manually (instead of using the `mc` command, use the command `mcseq` which is identical to `mc` except that it can perform model checking of sequential programs — these are programs with only one process which eventually terminates).

Q4: Is the program in Fig. 21 correct with respect to the last specification which was tried? Why? Check with TLV.

```
x : [0..4] where x = 1 ;

P1::[
  l_0: loop forever do [
    l_1: skip;
    l_2: x := x + 1;
    l_3: if x = 4 then l_4: x := 0;
   ];
  l_5:
]
```

Figure 22: Program mod4.spl

```
x : [0..5] where x = 1 ;

P1::[
  l_0: loop forever do [
    l_1: skip;
    l_2: x := x + 1;
    l_3: if x = 5 then l_4: x := 0;
   ];
  l_5:
]
```

Figure 23: Program mod5.spl

References

[1] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[2] K. L. McMillan. *The SMV System Draft.* Carnegie-Mellon Univerisity, 1992.

[3] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[4] Z. Manna and R. Waldinger. Is 'sometime' sometimes better than 'always'?: Intermittent assertions in proving program correctness. *Comm. ACM*, 21:159–172, 1978.

[5] Armin Biere, Alessandro Cimatti, Edmond Clarke, and Zhu Yunshan. Symbolic model checking without bdds. In *To be published*, 1999.