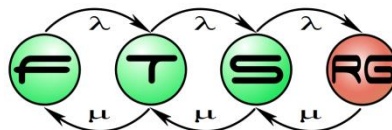


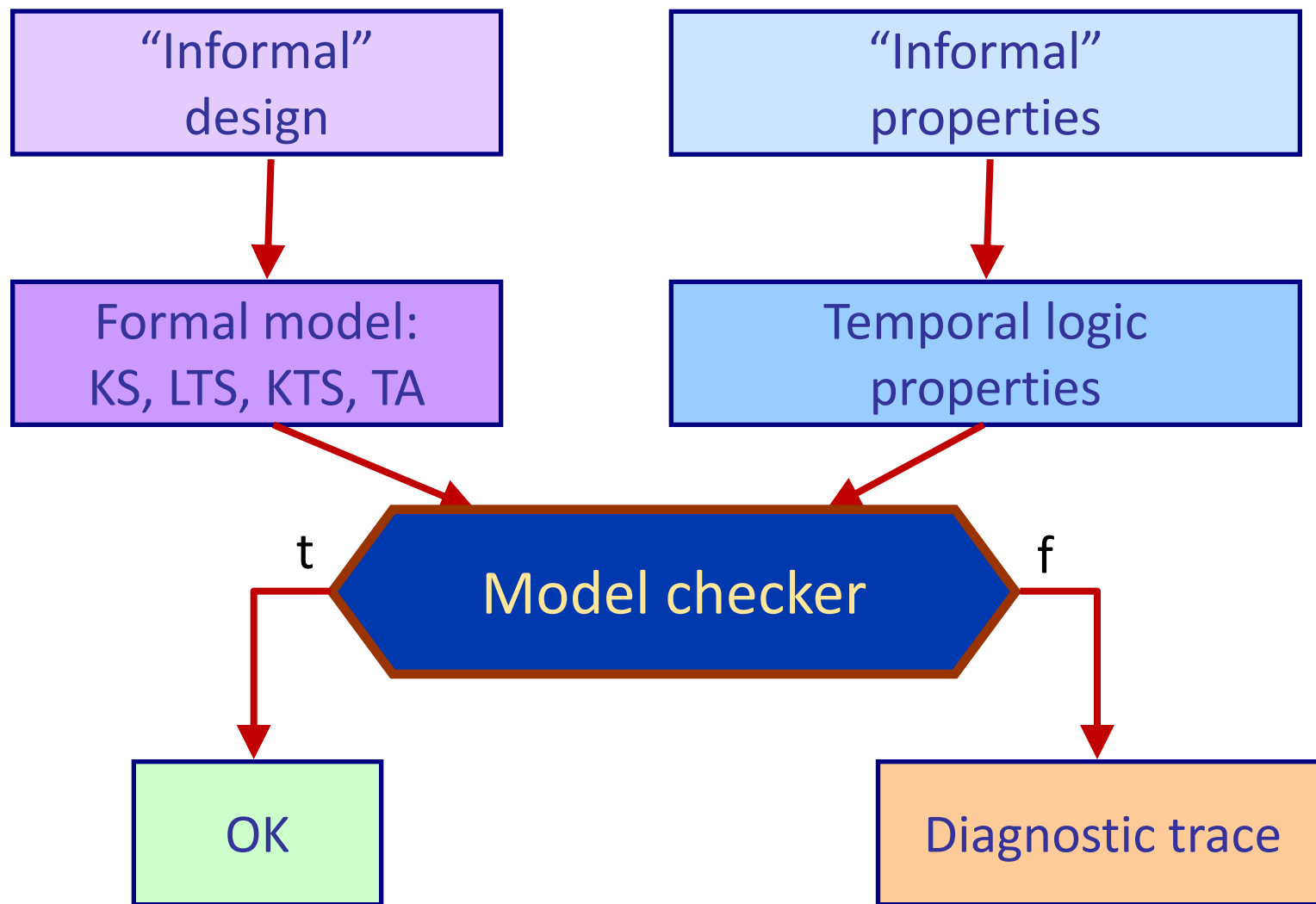
Model checker tools

István Majzik
<majzik@mit.bme.hu>

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group



Model checking as formal verification



Classic model checker tools

Tool	Models	Checked property	Recommended use
UPPAAL uppaal.org	Network of Extended Timed Automata	Restricted CTL (with clock variables)	Verification of time dependent behavior , synchronous communication
SPIN spinroot.com	Process Meta Language (Promela)	LTL, labels, property automaton (never claim)	Protocols and algorithms of asynchronous processes communicating using message queues
NuSMV nusmv.fbk.eu	Synchronous and asynchronous finite state machines	CTL, LTL	Algorithms of processes with shared variables , synchronous hardware components

The SPIN model checker and basics of its Promela language

The modeling language in SPIN

Promela: Process Meta Language

- **Processes:** Units of concurrent execution
 - Components in distributed algorithms or protocols
 - Nondeterministic behavior can be specified
- **Channels:** For interactions among processes
 - Asynchronous: FIFO message queue with given length
 - Synchronous: rendezvous, handshake
- **Variables**
 - Local variables in processes
 - Global (shared) variables among processes

Data types

■ Basic data types:

- **bool** or **bit** (1 bit), **byte** (8 bits),
short (16 bits, signed), **int** (32 bits, signed)
- Enumeration: **mtype** = {control, data, error}

■ Channels

- **chan** name = [length] of {types} <- message: n-tuple
 - Example: **chan c = [5] of {bit, int}**
- **Buffered** (asynchronous, FIFO), if length is not 0
- **Not buffered** (synchronous), if length is 0

■ Structured types

- Arrays: **int x[10]; chan c[3] = [6] of {bit, int, chan};**
- Records: **typedef MSG {bit control[5]; int data}**
- Using records: **MSG m, m.control[3], m.data**

Processes

- Definition („process type“):
`proctype procname (formal_parameters) {local_declarations; statements}`
- Instantiation
 - `init` process: default process that starts at the beginning
 - `active [num]` definition before `proctype`: automatic start
 - `run` statement: starting a process, e.g., `run A()`
 - Process parameters: data of basic type, channel
- Statements
 - Side-effect free expression is allowed
 - Separation of statements with `;` or `->` (equivalent)

```
byte state = 2;  
proctype A( ) {  
    (state == 1) -> state = 3  
}
```

```
byte state = 2;  
proctype A( ) {  
    state == 1;  
    state = 3  
}
```

Execution of statements

- A statement is either **executable** or **blocked**
 - Execution “gets stuck” on a blocked statement (until it becomes executable)
 - If a statement is executable then it can be executed
- Empty statement: **skip**
 - Always executable
- Assignment: e.g., **$x=x-1$**
 - Always executable
- Expression (condition)
 - Executable, if its evaluation is not 0 (false)
 - E.g., **$(a == b)$** is blocked if **$a \neq b$**
- Unconditional jump: **goto label** to a statement with **label**:
 - Always executable
- Timeout: **timeout**
 - Executable, if there is no other executable statement

Selection

■ Syntax:

if

:: statements

...

:: statements

:: **else** statements

fi

```
if
```

```
:: count = count+1
```

```
:: count = count-1
```

```
fi
```

■ Execution:

- The statements starting with **::** are called “options”
- An option is executable if its first statement is executable
- Option with **:: else** is executable only if other option isn't
- In case of many executable options: there is random selection
- Selection structure is executable if at least one option is executable

Repetition

■ Syntax:

do

:: statements

...

:: statements

:: **else** statements

od

```
do
```

```
:: count = count + 1;
```

```
:: count = count - 1;
```

```
:: (count == 0) -> break
```

```
od
```

■ Execution>

- The repetition is executable if at least one option is executable (i.e., the first statement of at least one option is executable)
- In case of many executable options: there is random selection
- Option with **:: else** is executable only if other option isn't
- After executing an option the repetition will start again
- Exit from the repetition: **break** or **goto label**

Examples for repetition and selection

```
proctype Euclid(int x, y) {  
    do  
        :: (x > y) -> x = x - y  
        :: (x < y) -> y = y - x  
        :: (x == y) -> goto done  
    od;  
done:  
    skip  
}
```

```
proctype counter() {  
    do  
        :: (count != 0) ->  
            if  
                :: count = count+1  
                :: count = count-1  
            fi  
        :: (count == 0) -> break  
    od  
}
```

Using channels

- Syntax of statements in case of channel q :
 - Sending: $q! e_1, e_2, \dots, e_n$ <- sending one message, variables or constants
 - Receiving: $q? e_1, e_2, \dots, e_n$ <- receiving one message, variables or constants
 - Checking: $\text{empty}(q)$, $\text{nempty}(q)$, $\text{full}(q)$, $\text{nfull}(q)$, $\text{len}(q)$
- Execution on **buffered** channel (FIFO, queue length is >0)
 - **Sending** is not executable if the channel is **full**, otherwise the sent message is put to the **tail** of the channel queue
 - **Receiving** is executable if the channel is **not empty**, and the specified constants **match the constants** of the message at the **head** of the channel
 - Constants are typically used to specify message type
 - When receiving a message, its values v_1, v_2, \dots become the values of the variables e_1, e_2, \dots specified in the receiving statement
- Execution on **not buffered** (synchronous) channel
 - Sending and receiving are executable together if these are **simultaneously executable** and the **constants** specified in their statements match
 - The written values will become the values of the variables specified in the receiving statement

Example for using a channel

```
chan Product[2] = [5] of {byte};
```

```
proctype Producer(byte pid) {  
    do  
        :: Product[pid] ! 1  
    od  
}
```

```
proctype Consumer( ) {  
    byte x;  
    do  
        :: Product[0] ? x;  
        :: Product[1] ? x  
    od  
}
```

```
init { run Producer(0); run Producer(1); run Consumer( ) }
```

Special expressions

■ **atomic** keyword

- Sequence is to be executed as one indivisible unit
atomic { (state==1) -> state = state + 1 }
- Not interleaved with any other processes
- Atomicity is lost in case of blocked internal statement

■ **d_step** keyword

- Similar to the **atomic** keyword
+ **deterministic internal execution** of statements
(even in case of random selection)
- Exiting or jumping to its internal statement is not allowed
- Blocking on an internal statement results in error

Further features

- See at: <http://spinroot.com/spin/Man/promela.html>
- Specific receive and send statements
 - `q? args` (normal)
 - `q?? args` (receiving from anywhere in the channel)
 - `q? <args>` (copying only)
 - `q?? <args>` (copying only, from anywhere)
 - `q? [args]` (polling)
 - `q?? [args]` (poling, from anywhere)
- Special constructs
 - `for(...), do ... od unless(...)`
 - `select`
 - `enabled`
 - `eval()`
 - ... and many more

General structure of a model

```
global_declarations;  
proctype procname1 (formal_parameters1) {  
    local_declarations1;  
    statements1  
};  
  
...  
proctype procnamen (formal_parametersn) {  
    local_declarationsn;  
    statementsn  
};  
  
init { ... run(procnamej) ... run(procnamek) ... }  
never { ... }
```


Mutual exclusion algorithm of Dekker

```
#define true  1
#define false 0
#define Aturn false
#define Bturn true

bool x, y, t;

proctype A() {
    x = true;
    t = Bturn;
    (y == false || t == Aturn);
    /* critical section */
    x = false
}
```

```
proctype B() {
    y = true;
    t = Aturn;
    (x == false || t == Bturn);
    /* critical section */
    y = false
}

init { run A(); run B(); }
```

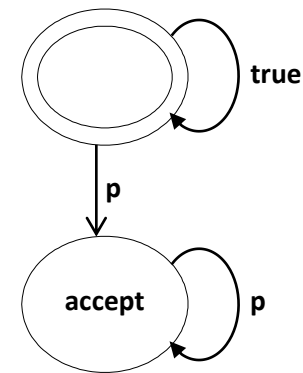
Specifying the properties to be verified

- **Assertions:** `assert()` condition, that shall be true
 - E.g., `assert(x!=y)`
- **Labels** on statements (incl. repetition, selection)
 - Acceptable end state: `end` prefix (e.g., `end`, `end1`, `end_a`)
 - To be executed for progress: `progress` prefix (i.e., infinite execution without progress can be checked)
- **never claim**
 - Specific process, consists of conditions only
 - If it matches with model execution then an error is detected
- **LTL temporal logic** (mapped to `never` claim)
 - Syntax: `ltl property_name {...}`
 - E.g., `ltl my_property {p U q}`
 - Operators: `U`, `W`, `F` denoted by `<>`, `G` denoted by `[]`, `X` is missing

Example for a never claim

- It is not allowed: Eventually, the property p becomes continuously true (i.e., $F G p$ is not allowed)

```
never { /* <>[]p */
  do
    :: true /* after an arbitrarily long prefix */
    :: p-> break /* p becomes true */
  od;
accept:
  do
    :: p /* and remains true forever after */
  od
}
```



- Specific label: **accept** prefix
 - If in the **never** claim the **accept** prefix is reachable infinitely often then an error is detected (match of the **never** claim)

Peterson mutual exclusion algorithm (assert)

```
bool turn, flag[2];           // the shared variables, booleans
byte ncrit;                  // nr of processes in critical section
```

```
active [2] proctype user() // two processes with built-in identifier _pid
{
```

```
    assert(_pid == 0 || _pid == 1);
```

```
again:
```

```
    flag[_pid] = 1;
```

```
    turn = _pid;
```

```
    (flag[1-_pid] == 0 || turn == 1-_pid);
```

```
    ncrit++;
```

```
    assert(ncrit == 1);
```

```
    ncrit--;
```

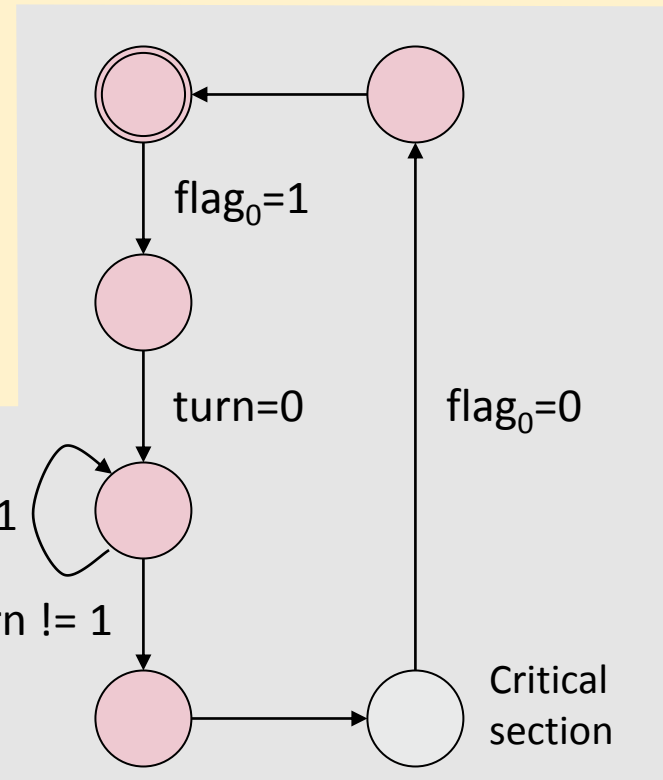
```
    flag[_pid] = 0;
```

```
    goto again
```

```
}
```

$flag_1 == 0 \ || \ turn == 1$

$flag_1 != 0 \ \&\& \ turn != 1$



Peterson mutual exclusion algorithm (LTL)

```
bool turn, flag[2];
bool critical[2];

active [2] proctype user()
{
    assert(_pid == 0 || __pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);

    critical[_pid] = 1;
    /* critical section */
    critical[_pid] = 0;

    flag[_pid] = 0;
    goto again;
}
```

LTL expressions:

[] (critical[0] || critical[1])

[] !(critical[0] && critical[1])

[] <> (critical[0])

[] <> (critical[1])

[] (critical[0] ->
(critical[0] U
(!critical[0] &&
(!critical[0] &&
!critical[1]) U critical[1])))

[] (critical[1] ->
(critical[1] U
(!critical[1] &&
(!critical[1] &&
!critical[0]) U critical[0])))

The SPIN model checker

- Command line tool
 - Several switches
- Eclipse RCP frame: SpinRCP
 - Model editor
 - Syntax checker
 - Automaton view
 - Simulation (with MSC visualization)
 - Verification with various parameters

The screenshot shows the 'Verification' dialog box in Eclipse RCP. It features a 'Verification Profile' section with 'MyVerificationProfile.xml' and buttons for 'Import', 'Export', and 'Reload'. Below are three tabs: 'Basic Options', 'Advanced Options', and 'Iterative/Swarm Run'. The 'Basic Options' tab is active, showing 'Correctness Properties' with radio buttons for 'Safety (state properties)', 'Liveness (cycles/sequences)', and 'Acceptance cycles', and checkboxes for 'Assertion violations', 'Invalid end states', 'Add weak fairness', 'Report unreachable code', and 'Check xr/xs assertions'. The 'Storage Mode' section has radio buttons for 'Exhaustive', 'Minimized automata', 'Bitstate hashing/Supertrace', and 'Hash-compact', along with checkboxes for 'Collapse compression'. The 'User Parameters' section has a checkbox for 'Use these parameters:' and input fields for 'Compile-time:' and 'Run-time:'. The 'Never Claim Specification' section has a checked checkbox for 'Apply never claim (if present) using:' and radio buttons for 'In-model LTL formula/claim name:' (with a text field containing 'p1'), 'LTL formula in the text field:', 'LTL formula in a 1-line file:', and 'Never claim in a file:', each with a 'Browse' button.

SpinRCP complete view

The screenshot displays the SpinRCP software interface, version 3.1.0, running on 30 December 2016. The interface is divided into several panes:

- Model Navigator:** Shows a tree view of project files, including examples, exercises, and various Petri net models like `bakery.pml` and `leader.pml`.
- Code Editor:** Displays the source code for `leader.pml`, which defines a Petri net with processes `p0` through `p3`, a number of processes `N`, and a number of locations `L`. It includes a `proctype` for `nnode` and a `chan` for `q[N]`.
- Simulation:** Shows a random simulation of the Petri net. The simulation step is 202. The simulation data values and queues are listed below.
- Simulation Data Values and Queues:**

Variable values	Queue contents values
<code>nnode(2):Active = 0</code>	<code>queue 1 (nnode(1):inp)</code>
<code>nnode(2):neighbourR = 1</code>	<code>queue 2 (nnode(2):inp)</code>
<code>nnode(2):nr = 5</code>	<code>queue 3 (nnode(3):inp)</code>
<code>nnode(3):Active = 0</code>	<code>queue 4 (nnode(4):inp)</code>
<code>nnode(3):neighbourR = 3</code>	<code>queue 5 (nnode(5):inp)</code>
<code>nnode(3):nr = 5</code>	
<code>nnode(4):Active = 0</code>	
<code>nnode(4):neighbourR = 4</code>	
<code>nnode(4):nr = 5</code>	
<code>nnode(5):know_winner =</code>	
<code>nnode(5):maximum = 5</code>	
<code>nnode(5):neighbourR = 5</code>	
<code>nnode(5):nr = 5</code>	
<code>nr_leaders = 1</code>	
- MSC Viewer:** Shows a Message Sequence Chart (MSC) for the simulation. The participants are `node2`, `node3`, `node4`, and `node5`. Messages are labeled `2winner:5`, `3winner:5`, `4winner:5`, and `5winner:5`.
- Console:** Shows the output of the simulation, including process creation and termination messages. The output indicates that 6 processes were created and all terminated successfully.
- Spin Trail to MSC:** Provides options for converting a Spin trail file to an MSC file. The spin trail file is `leader.pmlrnd_sim.out` and the MSC file is `leader.pmlrnd_sim.out`.

The NuSMV model checker

The modeling language in NuSMV (1)

- Finite State machine (FSM) with variables
 - Defining states and "possible next state" relation among the states
 - Variable with types: boolean, integer, enum, array
- Declaration of variables:
 - **VAR** section in the model: `identifier : type;`
- Initial state of the FSM: Initial assignments
 - **ASSIGN** section in the model: `init(identifier) := simple_expression;`
 - Variable without assignment: `input` (any value assigned according to its type)
- Next state transition in the FSM: Changing the values of variables
 - **ASSIGN** section: `next(identifier) := next_expression;`
the expression may refer to the value of variables in the current and in the next state (the latter with the `next()` operator);
next_expression may contain set of values to choose from randomly
 - **ASSIGN** section: `identifier := simple_expression;`
defines the value of the variable for all states

The modeling language in NuSMV (2)

- Conditional expressions

- **if-then-else expression** according to the usual (C-like) syntax

`condition ? expression1: expression2`

- **case expression**: the first option with a true condition determines the expression to be executed (error if there is no true option or TRUE option)

`case`

`condition1 : expression1;`

`...`

`conditionn : expressionn;`

`TRUE: expressiondefault;`

`esac`

- Assignment to variables with **constraints** (logic expressions)

- **INIT** section: Any initial value which satisfies the constraint
- **TRANS** section: Current and next values (see the `next()` operator) shall satisfy the constraint

Example model: Producer

```
MODULE main
```

```
VAR
```

```
  request: boolean;
```

```
  state: {ready, busy};
```

```
ASSIGN
```

```
  init(state) := ready;
```

```
  next(state) :=
```

```
    case
```

```
      state = ready & request: busy;
```

```
      state = busy: {ready, busy};
```

```
      TRUE: ready;
```

```
    esac;
```

The modeling language in NuSMV (3)

- `if()` condition
if ($x < S$ & $b > 0$)
 `next(x) := x+1`
- `for(; ;)` loop
for ($j=1$; $j \leq N-1$; $j=j+1$)
 `next(a[j] := a[j-1])`

The property description in NuSMV

- **Invariants**
 - **INVAR** section: logic expression for values of variables
- **CTL expressions**
 - **CTLSPEC** or **SPEC** section, standard notation
 - Logic expressions instead of atomic propositions
 - E.g.: **CTLSPEC** $AG(\text{request} \rightarrow AF(\text{state} = \text{busy}))$
- **LTL expressions**
 - **LTLSPEC** section, standard notation (implicit A)
 - Logic expressions instead of atomic propositions
 - E.g.: **LTLSPEC** $G (y=4 \rightarrow X y=6)$
- **Useful: Alias (macro) definitions for propositions**
 - **DEFINE** section: $\text{alias} := \text{simple_expression}$

Modular structure

- Basic unit:
 - **MODULE** name, with (optional) parameter
 - E.g., **MODULE user(semaphore)**
- Processes instantiated from modules
 - **process** keyword in the **VAR** section
 - E.g.:
`proc1 : process user(semaphore);`
`proc2 : process user(semaphore);`
 - (This possibility may not be supported in the future)
- Specifying fair behavior
 - **FAIRNESS** section: **running** keyword,
or a CTL state expression that shall hold infinitely often
 - E.g.: **FAIRNESS running** (process runs infinitely often)

Semantics: Synchronous or asynchronous

- Synchronous execution
 - Instantiation of modules
 - In a "step" **each module** performs a state transition (assigning new values to some variables)
 - Preferred for verification of **hardware component**
- Asynchronous execution
 - Instantiation of modules with the **process** keyword in the main module
 - In a "step" only **one randomly selected module** performs a state transition (assigning new values to some variables)
 - Preferred for verification of **distributed systems** that use shared variables

Example: Synchronous or asynchronous system

```
MODULE cell(input)
  VAR
    val : {red, green, blue};
  ASSIGN
    next(val) := {input};
```

```
MODULE main
  VAR
    c1 : cell(c3.val);
    c2 : cell(c1.val);
    c3 : cell(c2.val);
```

```
MODULE cell(input)
  VAR
    val : {red, green, blue};
  ASSIGN
    next(val) := {input};
```

```
MODULE main
  VAR
    c1 : process cell(c3.val);
    c2 : process cell(c1.val);
    c3 : process cell(c2.val);
```


Example: Asynchronous system

```
MODULE main
```

```
VAR
```

```
semaphore : boolean;
```

```
proc1 : process user(semaphore);
```

```
proc2 : process user(semaphore);
```

```
ASSIGN
```

```
init(semaphore) := FALSE;
```

```
CTLSPEC AG ! (proc1.state = critical &  
proc2.state = critical)
```

```
CTLSPEC AG (proc1.state = entering ->  
AF proc1.state = critical)
```

```
LTLSPEC G ! (proc1.state = critical &  
proc2.state = critical)
```

```
LTLSPEC G (proc1.state = entering ->  
F proc1.state = critical)
```

```
MODULE user(semaphore)
```

```
VAR
```

```
state : {idle, entering, critical, exiting};
```

```
ASSIGN
```

```
init(state) := idle;
```

```
next(state) :=
```

```
case
```

```
state = idle : {idle, entering};
```

```
state = entering & !semaphore : critical;
```

```
state = critical : {critical, exiting};
```

```
state = exiting : idle;
```

```
TRUE : state;
```

```
esac;
```

```
next(semaphore) :=
```

```
case
```

```
state = entering : TRUE;
```

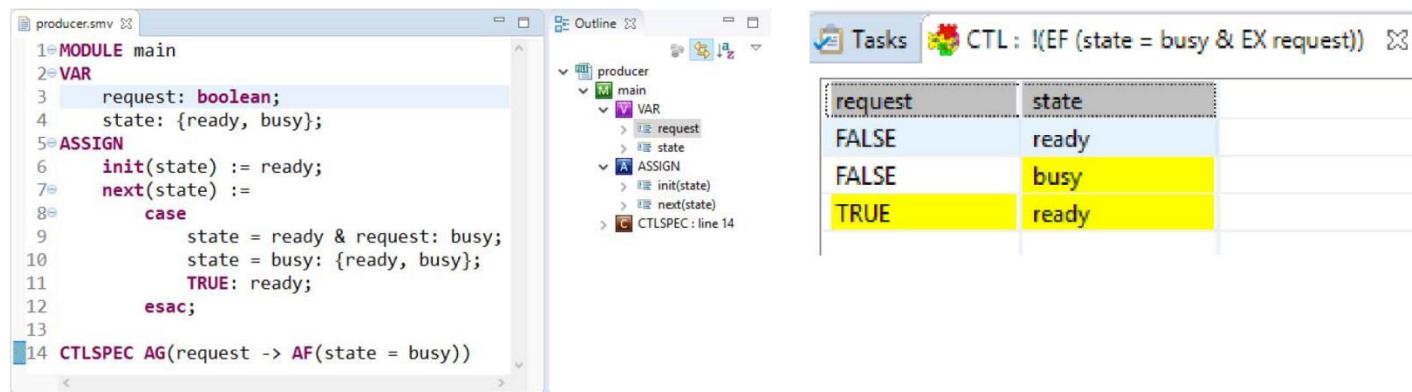
```
state = exiting : FALSE;
```

```
TRUE : semaphore;
```

```
esac;
```

The NuSMV model checker

- Command line version
 - Execution: `nusmv model`
 - Textual output
 - Counterexample is also textual (value of variables)
- Eclipse framework: NuSeen
 - Xtext based model editor
 - Tabular visualization of counterexamples
 - Dependency graphs of variables



```
1=MODULE main
2=VAR
3   request: boolean;
4   state: {ready, busy};
5=ASSIGN
6   init(state) := ready;
7   next(state) :=
8     case
9       state = ready & request: busy;
10      state = busy: {ready, busy};
11      TRUE: ready;
12    esac;
13
14 CTLSPEC AG(request -> AF(state = busy))
```

Tasks CTL: !(EF (state = busy & EX request))

request	state
FALSE	ready
FALSE	busy
TRUE	ready