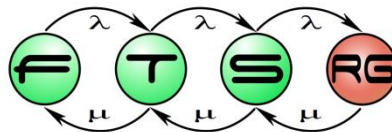


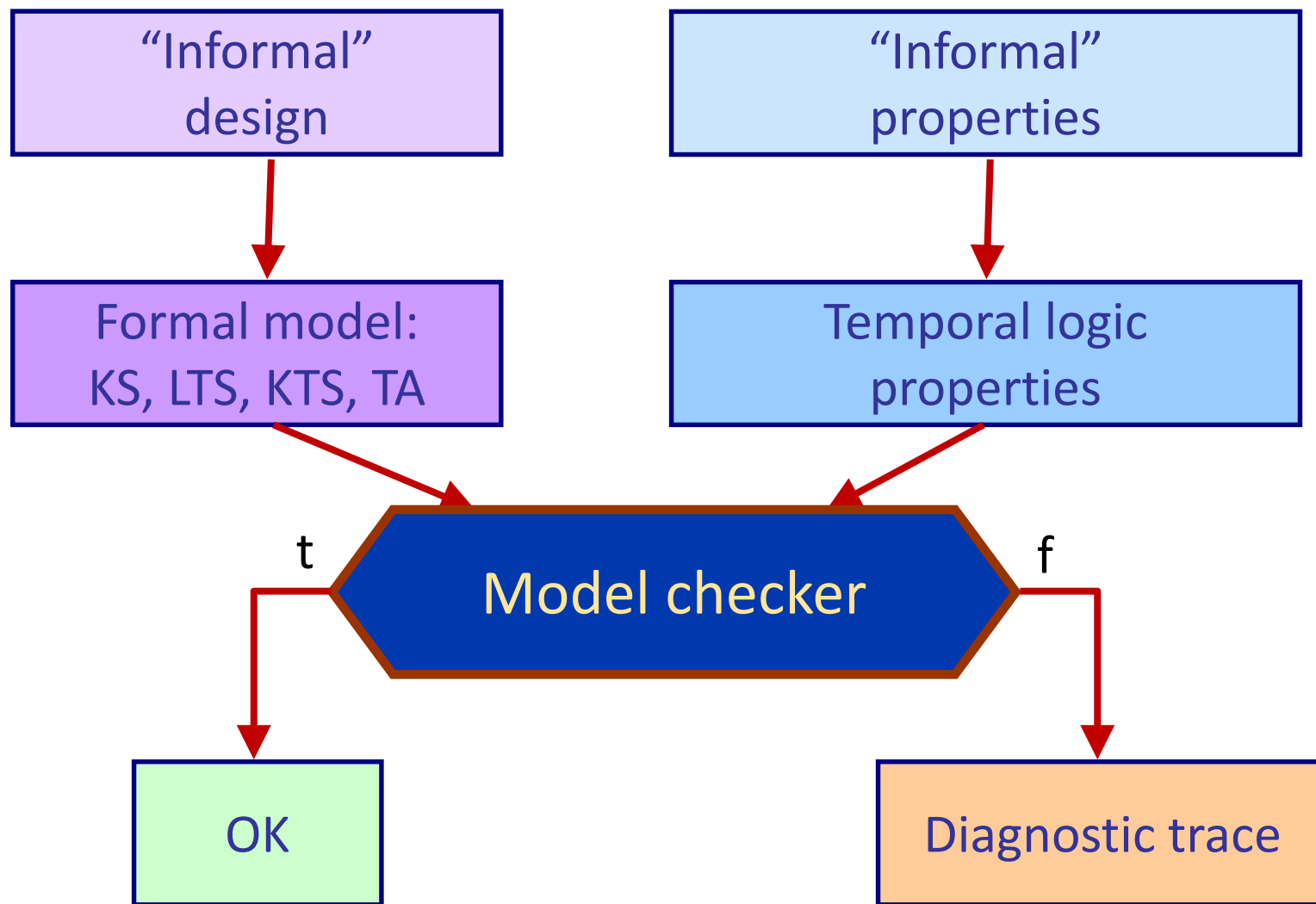
Model Checker Tools

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 the 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 = {...}`, e.g.: `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 of process template (“process type”):

```
proctype procname (formal_parameters) {local_declarations; statements}
```

- Instantiation

- **init** process: default process that starts initially
- **run** statement: starting a process instance, e.g., **run A()**
- **active [num]** before **proctype**: automatic start of **num** instances
- Possible process parameters: data of basic type, channel

- Statements

- Side-effect-free expression is also 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 **$a != b$**
- 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
- The **if ... fi** 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 a channel: $empty(q), nempty(q), full(q), nfull(q), len(q)$
- 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, the variables e_1, e_2, \dots specified in the receiving statement get assigned the values v_1, v_2, \dots of the received message
- 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 variables in the receiving statement will get assigned the values of the sent message

Example for using a channel

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

```
proctype Producer(byte mypid) {  
    do  
        :: Product[mypid] ! 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

- Statement sequence is to be executed as one indivisible unit, e.g.: **atomic { (state==1) -> state = state + 1 }**
- Not interleaved with any other process
- 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]` (polling, 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(); }
```

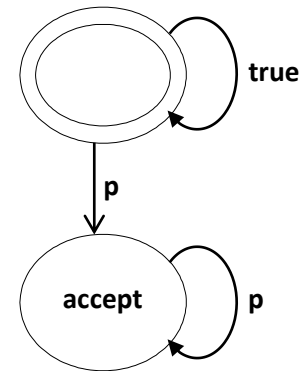
Specifying the properties to be verified

- **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 internally 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 that 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++;
```

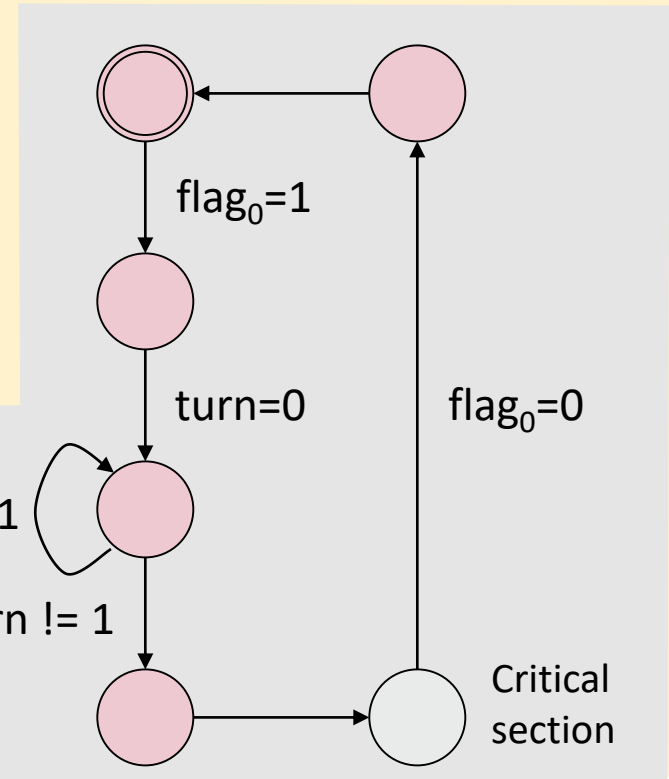
```
    assert(ncrit == 1);
```

```
    ncrit--;
```

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

```
    goto again
```

```
}
```



Peterson mutual exclusion algorithm (LTL)

```
bool turn, flag[2]; // Shared variables
bool critical[2]; // Being in the crit. sect.

active [2] proctype user() // built-in _pid
{
    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])
```

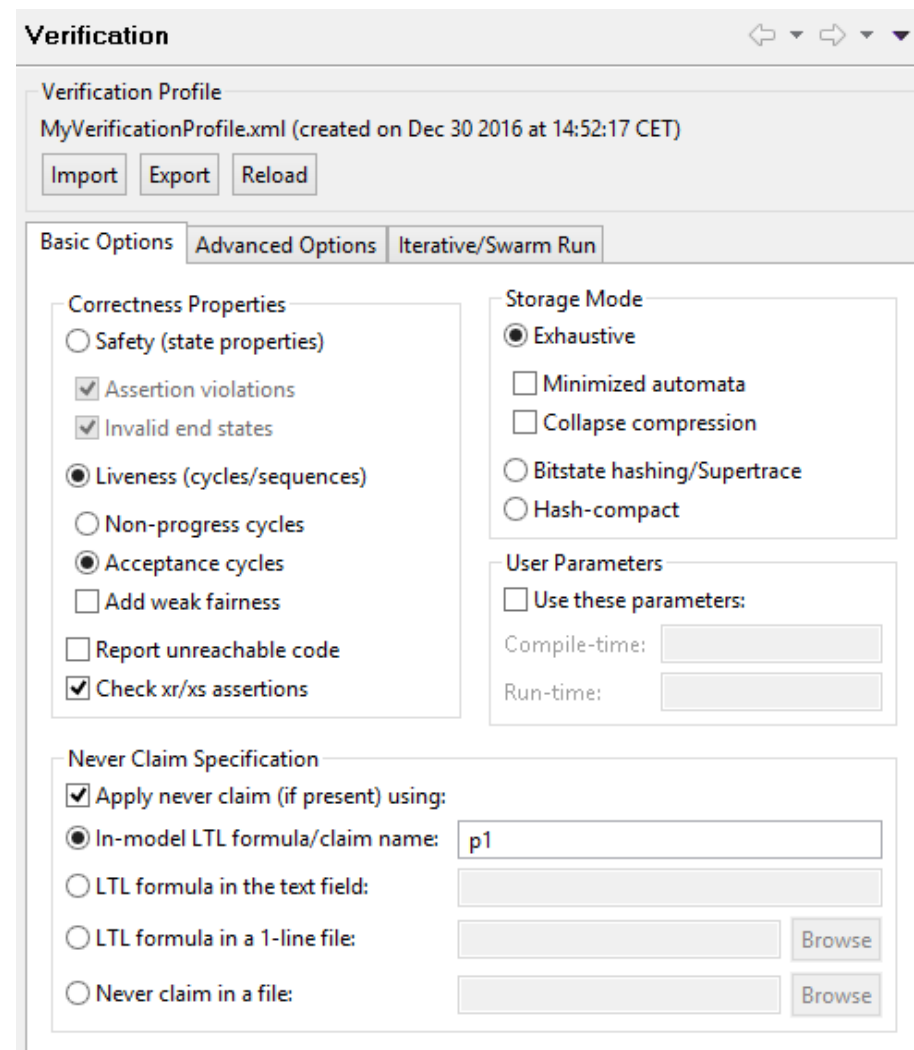
```
[] (critical[0] || critical[1])
```

```
[] (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



SpinRCP complete view

SpinRCP Version 3.1.0 -- 30 December 2016 :: Spin Version 6.4.6 -- 2 December 2016

File Edit Run Window Help

Save Save All Print... Syntax Check Redundancy Check Symbol Table Close Verification Automata View State Tables Export to MSC Import Trail

Model Navigator

- Examples
- Exercises
- Holzmann_1991
- Holzmann_2003
- RJA
- LTL
 - _spin_nvt.nmp
 - bakery.ltl
 - bakery.pml
 - bakery.pml.gui_sim.out
 - bakery.pml.md_sim.out
 - bakery.pml.trail
 - bakery.prp
 - diskhead-automata
 - diskhead-automata.2.p
 - diskhead-automata.pd
 - diskhead.pml
 - diskhead.pml.nvt
 - diskhead.pml.md_sim
 - leader-automata
 - leader-automata.2.pdf
 - leader-automata.2.svg
 - leader-automata.3.pdf
 - leader-automata.3.svg
 - leader-automata.4.pdf
 - leader-automata.4.svg
 - leader-automata.5.pdf
 - leader-automata.5.svg
 - leader-automata.6.pdf
 - leader-automata.6.svg
 - leader-automata.dcf
 - leader-automata.pdf
 - leader-automata.svg
 - leader.pml
 - leader.pml.ltl_sim.out
 - leader.pml.ltl
 - leader.pml.md_sim.ms
 - leader.pml.md_sim.out
 - leader.pml.trail
 - ltl_always_eventually-a
 - ltl_always_eventually-a
 - ltl_always_eventually.d
 - ltl_always_eventually.dl

leader.pml

```

26 byte nr_leaders = 0;
27
28 ltl p0 { <> (nr_leaders > 0) }
29 ltl p1 { <> [] (nr_leaders == 1) }
30 ltl p2 { [] (nr_leaders == 0 U nr_l
31 ltl p3 { ![] (nr_leaders == 0) }
32
33 #define N 5 /* number of processe
34 #define L 10 /* 2xN */
35 byte I;
36
37 wtype = { one, two, winner };
38 chan q[N] = [L] of { wtype, byte };
39
40 proctype nnode (chan inp, out; byte
41 {
42   bit Active = 1, know_winner = 0;
43   byte nr, maximum = nnumber, nei
44   xr Imp;
45   xs out;

```

MSC Viewer

```

sequenceDiagram
    participant nnode2
    participant nnode3
    participant nnode4
    participant nnode5
    nnode2->>nnode5: 2winner:5
    nnode3->>nnode5: 3winner:5
    nnode4->>nnode5: 4winner:5
    nnode5->>nnode5: 5winner:5

```

Simulation

Simulation / Replay

Random simulation: leader.pml

Run Stop Message Step Single Step

Simulation View Options

Ignore selected channels:

Follow selected channels:

Rename selected channels:

Create virtual processes:

Hide message parameters

Simulation Data Values and Queues

Simulation step: 202

Variable values

```

nnode(2):Active = 0
nnode(2):neighbourR = 1
nnode(2):nr = 5
nnode(3):Active = 0
nnode(3):neighbourR = 3
nnode(3):nr = 5
nnode(4):Active = 0
nnode(4):neighbourR = 4
nnode(4):nr = 5
nnode(5):know_winner =
nnode(5):maximum = 5
nnode(5):neighbourR = 5
nnode(5):nr = 5
nr_leaders = 1

```

Queue contents values

```

queue 1 (nnode(1):inp):
queue 2 (nnode(2):inp):
queue 3 (nnode(3):inp):
queue 4 (nnode(4):inp):
queue 5 (nnode(5):inp):

```

Console

```

Spin [Random Simulation] C:\Users\Zmago\SpinRCP\workspace\LTL\leader.pml
197: proc 4 (nnode:1) leader.pml:91 (state 44)
MSC: LOST
198: proc 5 (nnode:1) leader.pml:81 (state 32)
199: proc 5 (nnode:1) leader.pml:87 (state 38)
200: proc 5 (nnode:1) leader.pml:98 (state 39)
201: proc 5 (nnode:1) leader.pml:91 (state 45)
202: proc 5 (nnode:1) leader.pml:91 (state 44)
202: proc 5 (nnode:1) terminates
202: proc 4 (nnode:1) terminates
202: proc 3 (nnode:1) terminates
202: proc 2 (nnode:1) terminates
202: proc 1 (nnode:1) terminates
202: proc 0 (:init:1) terminates
6 processes created
Random simulation trail written to leader.pml.rnd

```

Spin Trail to MSC

Conversion

Spin trail file: leader.pml.md_sim.out

MSC file:

Read/Convert:

Options

Rename selected channels: 1>m1 2>m2 3>m3 4>m4 5>m5

Create virtual processes: 2,3>joined23

Comment: MSC of leader.pml random simulation

Hide message parameters

Process and Message List

Process name	Process ID	Channel ID	Message	From	To
init:	0	1	winner:5	5	1
nnode	1	2	winner:5	1	2
nnode	2	3	winner:5	2	3
nnode	3	4	winner:5	3	4
nnode	4	5	winner:5	4	5

Writable Insert 26:1 219M of 308M

CVS Repositories

- pspserver:zmago@localhost:/CVS_Repository
 - HEAD
 - Branches
 - Versions
 - Data

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 variable` (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);
the `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 branch)
`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<=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, with standard notation
 - Logic expressions instead of atomic propositions
 - E.g.: **CTLSPEC** $AG(\text{request} \rightarrow AF(\text{state} = \text{busy}))$
- **LTL expressions**
 - **LTLSPEC** section, with 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** means: 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 components**
- 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: $\neg(EF (state = busy \ \& \ EX \ request))$

request	state
FALSE	ready
FALSE	busy
TRUE	ready