NuSMV: Advanced Features

– Symbolic Model Checking –

A. Cimatti and M. Pistore

ESSLLI, July 5-9, 2002, Trento (Italy)

The `DEFINE` declaration

In the following example, the values of variables `out` and `done` are defined by the values of the other variables in the model.

```
MODULE main          -- counter_8
VAR
  b0   : boolean;
  b1   : boolean;
  b2   : boolean;
  out  : 0..8;
  done : boolean;

ASSIGN
  init(b0) := 0;
  init(b1) := 0;
  init(b2) := 0;

  next(b0) := !b0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
  next(b2) := ((b0 & b1) & !b2) | (!(b0 & b1) & b2);

  out  := b0 + 2*b1 + 4*b2;
  done := b0 & b1 & b2;
```

The `DEFINE` declaration

`DEFINE` declarations can be used to define *abbreviations*:

```
MODULE main          -- counter_8
VAR
  b0 : boolean;
  b1 : boolean;
  b2 : boolean;

ASSIGN
  init(b0) := 0;
  init(b1) := 0;
  init(b2) := 0;

  next(b0) := !b0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
  next(b2) := ((b0 & b1) & !b2) | (!(b0 & b1) & b2);

DEFINE
  out  := b0 + 2*b1 + 4*b2;
  done := b0 & b1 & b2;
```

The `DEFINE` declaration

- The syntax of `DEFINE` declarations is the following:

```
DEFINE <id> := <simple_expression> ;
```

- They are similar to macro definitions.
- No new state variable is created for defined symbols (hence, no added complexity to model checking).
- Each occurrence of a defined symbol is replaced with the body of the definition.

Arrays

The SMV language provides also the possibility to define *arrays*.

```
VAR
  x : array 0..10 of booleans;
  y : array 2..4 of 0..10;
  z : array 0..10 of array 0..5 of {red, green, orange};
ASSIGN
  init(x[5]) := 1;
  init(y[2]) := {0,2,4,6,8,10};
  init(z[3][2]) := {green, orange};
```

☞ Remark: Array indexes in SMV *must be constants*.

Records

Records can be defined as modules without parameters and assignments.

```
MODULE point
  VAR x: -10..10;
      y: -10..10;

MODULE circle
  VAR center: point;
      radius: 0..10;

MODULE main
  VAR c: circle;
  ASSIGN
    init(c.center.x) := 0;
    init(c.center.y) := 0;
    init(c.radius)   := 5;
```

The constraint style of model specification

The following SMV program:

```
MODULE main
VAR request : boolean;
    state   : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
      state = ready & request : busy;
      1                       : {ready,busy};
  esac;
```

can be alternatively defined in a *constraint style*, as follows:

```
MODULE main
VAR request : boolean;
    state   : {ready,busy};
INIT
  state = ready
TRANS
  (state = ready & request) -> next(state) = busy
```

The constraint style of model specification

- The SMV language allows for specifying the model by defining constraints on:
  - the *states*:
    `INVAR <simple_expression>`
  - the *initial states*:
    `INIT <simple_expression>`
  - the *transitions*:
    `TRANS <next_expression>`
- There can be zero, one, or more constraints in each module, and constraints can be mixed with assignments.
- Any propositional formula is allowed in constraints.
- Very useful for writing translators from other languages to NuSMV.
- `INVAR p` is equivalent to `INIT p` and `TRANS next(p)`, but is more efficient.
- Risk of defining *inconsistent models* (`INIT p & !p`).

Assignments versus constraints

- Any `ASSIGN`-based specification can be easily rewritten as an equivalent constraint-based specification:

```
ASSIGN
  init(state) := {ready,busy};        INIT  state in {ready,busy}
  next(state) := ready;                TRANS next(state) = ready
  out := b0 + 2*b1;                    INVAR out = b0 + 2*b1
```

- The converse is not true: constraint

```
TRANS
  next(b0) + 2*next(b1) + 4*next(b2) =
                              (b0 + 2*b1 + 4*b2 + tick) mod 8
```

cannot be easily rewritten in terms of `ASSIGN`s.

Assignments versus constraints

☞ Models written in **assignment style**:

- by construction, there is always *at least one initial state*;
- by construction, all states have *at least one next state*;
- *non-determinism is apparent* (unassigned variables, set assignments...).

☞ Models written in **constraint style**:

- `INIT` constraints *can be inconsistent*:
  - inconsistent model: no initial state,
  - any specification (also `SPEC 0`) is vacuously true.
- TRANS constraints *can be inconsistent*:
  - the transition relation is not total (there are deadlock states),
  - NuSMV detects and reports this case.
- *non-determinism is hidden* in the constraints:

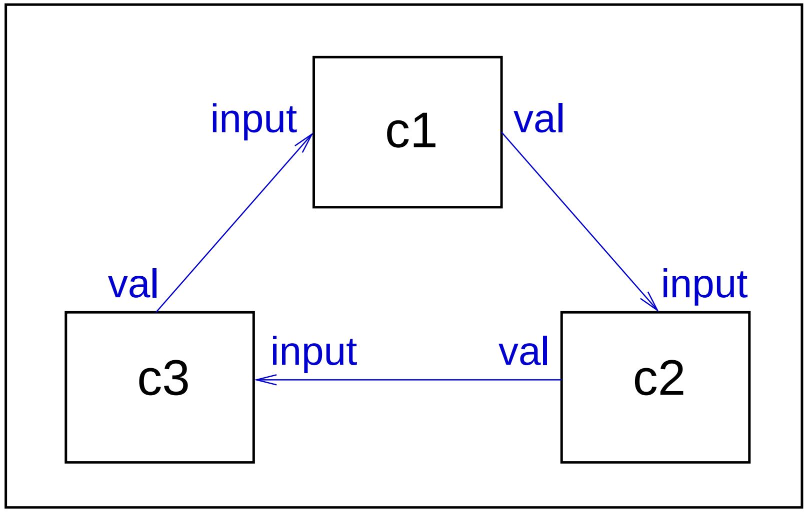
```
TRANS (state = ready & request) -> next(state) = busy
```

Synchronous composition

☞ By default, composition of modules is **synchronous**:
all modules move at each step.

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

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



Synchronous composition

A possible execution:

step	*c1.val*	*c2.val*	*c3.val*
0	**red**	**green**	**blue**
1	red	**red**	**green**
2	**green**	red	green
3	green	red	green
4	green	red	**red**
5	**red**	**green**	red
6	red	**red**	red
7	red	red	red
8	red	red	red
9	red	red	red
10	red	red	red

Asynchronous composition

- **Asynchronous** composition can be obtained using keyword `process`.
- In asynchronous composition *one process moves at each step*.
- Boolean variable `running` is defined in each process:
  - it is true when that process is selected;
  - it can be used to guarantee a fair scheduling of processes.

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

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

Asynchronous composition

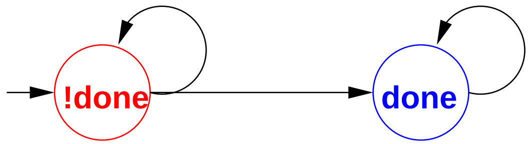
A possible execution:

step	*runnig*	*c1.val*	*c2.val*	*c3.val*
0	-	**red**	**green**	**blue**
1	c2	red	**red**	blue
2	c1	**blue**	red	blue
3	c1	blue	red	blue
4	c2	blue	red	blue
5	c3	blue	red	**red**
6	c2	blue	**blue**	red
7	c1	blue	blue	red
8	c1	**red**	blue	red
9	c3	red	blue	**blue**
10	c3	red	blue	blue

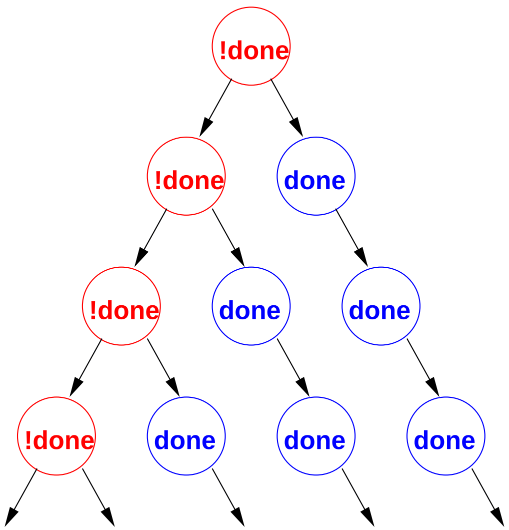
Paths and trees

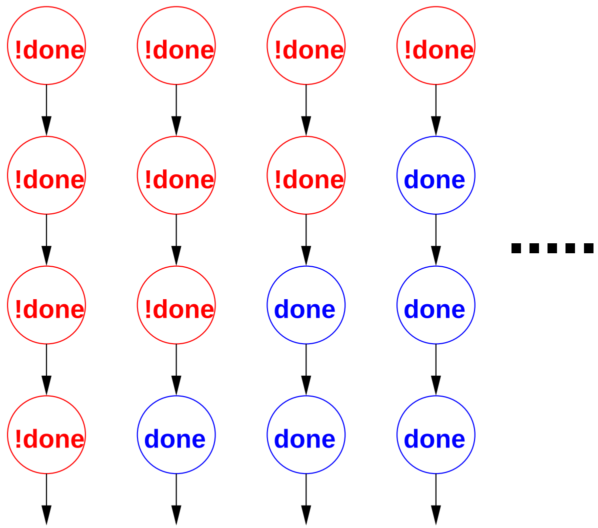
- An SMV specification defines a Kripke structure:

```
MODULE main
VAR done: boolean;
INIT  !done
TRANS done -> next(done)
```



- The execution of the Kripke structure can be seen as:
  - an infinite *tree*
  - as a set of infinite *paths*.





Specifications

In the SMV language:

☞ Specifications can be added in any module of the program.

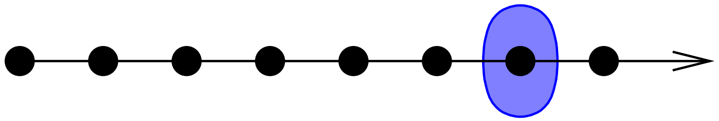
☞ Each property is verified separately.

☞ Different kinds of properties are allowed:

- Properties on the reachable states
  - *invariants* (`INVARSPEC`)
- Properties on the computation paths (*linear time* logics):
  - LTL (`LTLSPEC`)
  - qualitative characteristics of models (`COMPUTE`)
- Properties on the computation tree (*branching time* logics):
  - CTL (`SPEC`)
  - Real-time CTL (`SPEC`)

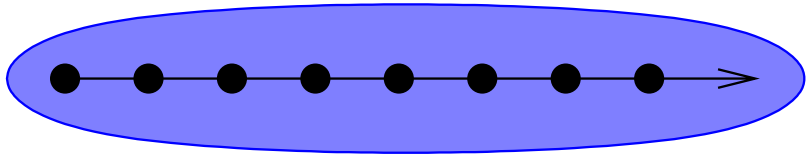
LTL specifications

finally p



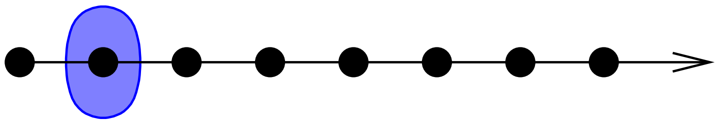
F p

globally p



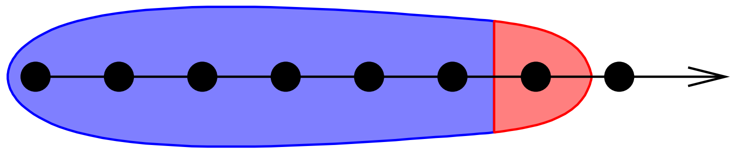
G p

next p



X p

p until q



p U q

LTL specifications

- ☞ LTL properties are specified via the keyword `LTLSPEC`:

```
LTLSPEC <ltl_expression>
```

- ☞ A state in which `out = 3` is eventually reached.

```
LTLSPEC F out = 3
```

- ☞ Condition `out = 0` holds until `reset` becomes false.

```
LTLSPEC (out = 0) U (!reset)
```

- ☞ Even time a state with `out = 2` is reached, a state with `out = 3` is reached afterwards.

```
LTLSPEC G (out = 2 -> F out = 3)
```

Quantitative characteristics computations

It is possible to compute the minimum and maximum length of the paths between two specified conditions.

- ☞ Quantitative characteristics are specified via the keyword `COMPUTE`:

```
COMPUTE MIN/MAX [ <simple_expression> , <simple_expression> ]
```

- ☞ For instance, the shortest path between a state in which `out = 0` and a state in which `out = 3` is computed with

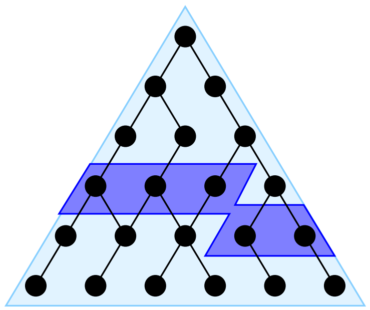
```
COMPUTE
  MIN [ out = 0 , out = 3]
```

- ☞ The length of the longest path between a state in which `out = 0` and a state in which `out = 3`.

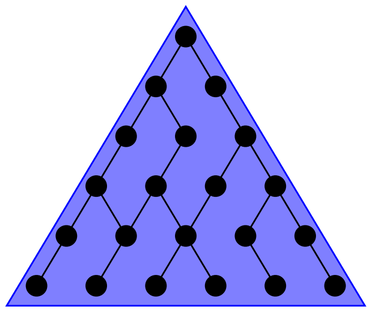
```
COMPUTE
  MAX [ out = 0 , out = 3]
```

CTL specifications

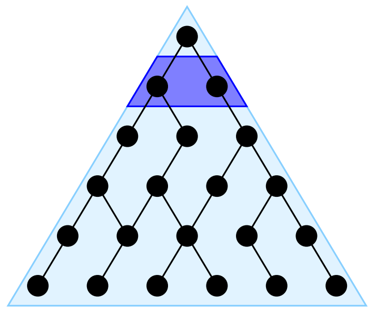
finally p | globally p | next p | p until q



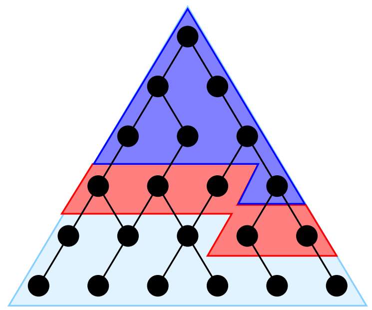
AF p



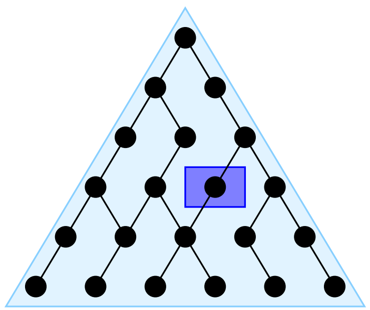
AG p



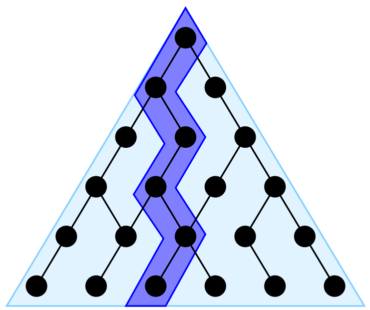
AX p



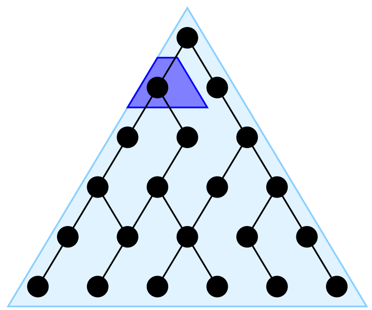
A[p U q]



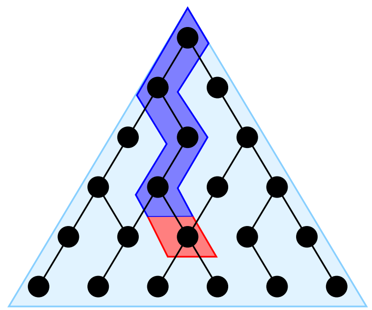
EF p



EG p



EX p



E[p U q]

CTL specifications

- CTL properties are specified via the keyword `SPEC`:

```
SPEC <ctl_expression>
```

- It is possible to reach a state in which `out = 3`.

```
SPEC EF out = 3
```

- A state in which `out = 3` is always reached.

```
SPEC AF out = 3
```

- It is always possible to reach a state in which `out = 3`.

```
SPEC AG EF out = 3
```

- Even time a state with `out = 2` is reached, a state with `out = 3` is reached afterwards.

```
SPEC AG (out = 2 -> AF out = 3)
```

Bounded CTL specifications

NuSMV provides *bounded CTL* (or *real-time CTL*) operators.

- ☞ There is no state that is reachable in 3 steps where `out = 3` holds.

```
SPEC
  !EBF 0..3 out = 3
```

- ☞ A state in which `out = 3` is reached in 2 steps.

```
SPEC
  ABF 0..2 out = 3
```

- ☞ From any reachable state, a state in which `out = 3` is reached in 3 steps.

```
SPEC
  AG ABF 0..3 out = 3
```

NuSMV resources

- NuSMV home page: `http://nusmv.irst.itc.it/`
- Mailing lists:
  - `nusmv-users@irst.itc.it` (public discussions)
  - `nusmv-announce@irst.itc.it` (announces of new releases)
  - `nusmv@irst.itc.it` (the development team)
  - to subscribe: `http://nusmv.irst.itc.it/mail.html`
- Course notes and slides:
  `http://nusmv.irst.itc.it/courses/esslli02/`
  (will be ready this evening...)