Model Checking

A Hands-On Introduction

A. Cimatti, M. Pistore, and M. Roveri

13^{th} International Conference on Automated Planning & Scheduling
June 10 2003
Trento (Italy)

Acknowledgement

These slides are derived from courses on NuSMV and Symbolic Model Checking (see http://nusmv.irst.itc.it/courses/).

The goals of the course are:

- to provide a practical introduction to symbolic model checking,
- to describe the basic feautures of the NuSMV symbolic model checker.

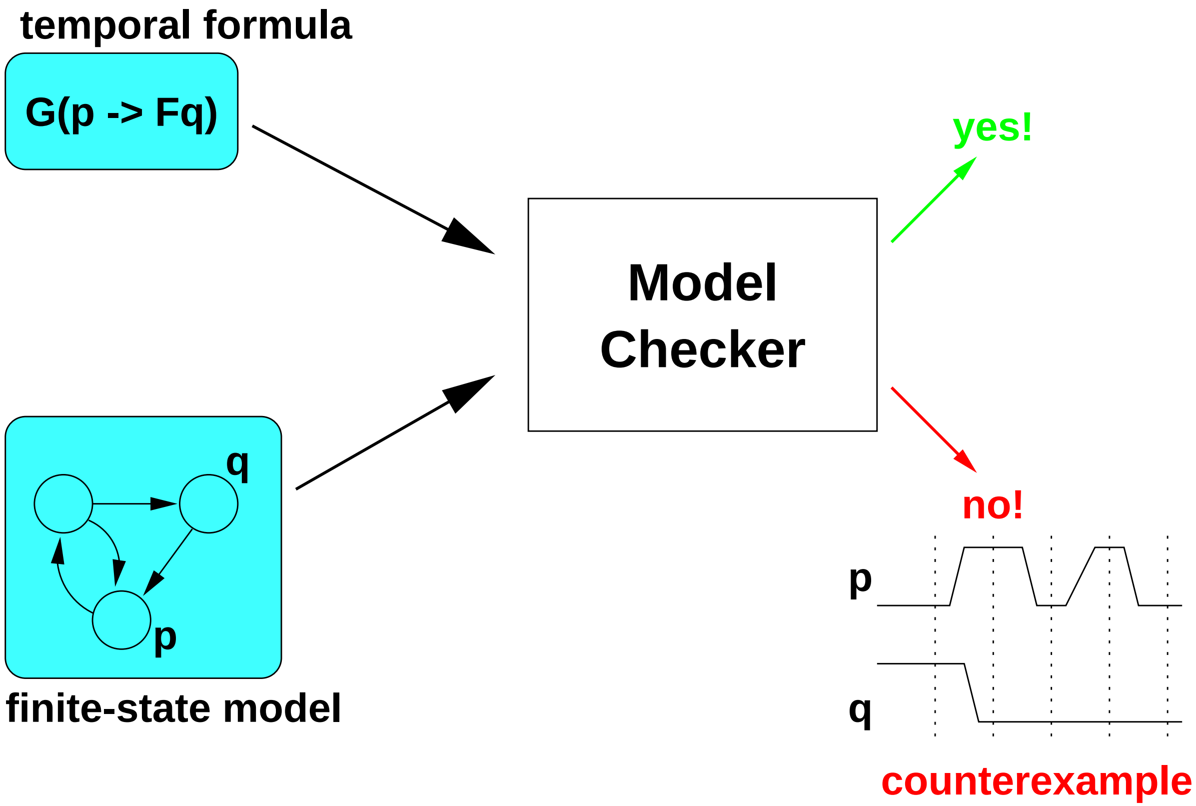
Authors of the slides:

- Alessandro Cimatti (ITC-irst)
- Marco Pistore (ITC-irst and University of Trento)
- Marco Roveri (ITC-irst)

Industrial success of Model Checking

- From academics to industry in a decade.
- Easier to integrate within industrial development cycle:
  - input from practical design languages (e.g. VHDL, SDL, StateCharts);
  - expressiveness limited but often sufficient in practice.
- Does not require deep training (“push-button” technology).
  - Easy to explain as exhaustive simulation.
- Powerful debugging capabilities:
  - detect costly problems in early developmemt stages (cfr. Pentium bug);
  - exhaustive, thus effective (often bugs are also in scaled-down problems).
  - provides counterexamples (directs the designer to the problem).

What is a Model Checker



Description languages for Kripke Models

A Kripke model is usually presented using a structured programming language.

Each component is presented by specifying

- state variables: determine the state space S and the labeling L.
- initial values for state variables: determine the set of initial states I.
- instructions: determine the transition relation R.

Components can be combined via

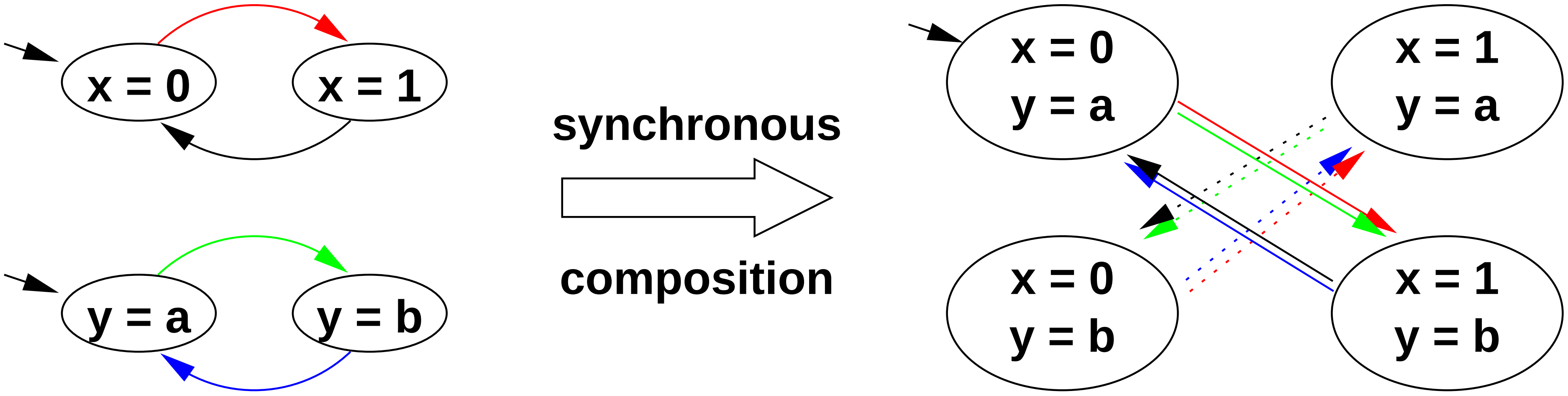
- synchronous composition,
- asynchronous composition.

State explosion problem in model checking:

- linear in model size, but model is exponential in number of components.

Synchronous Composition

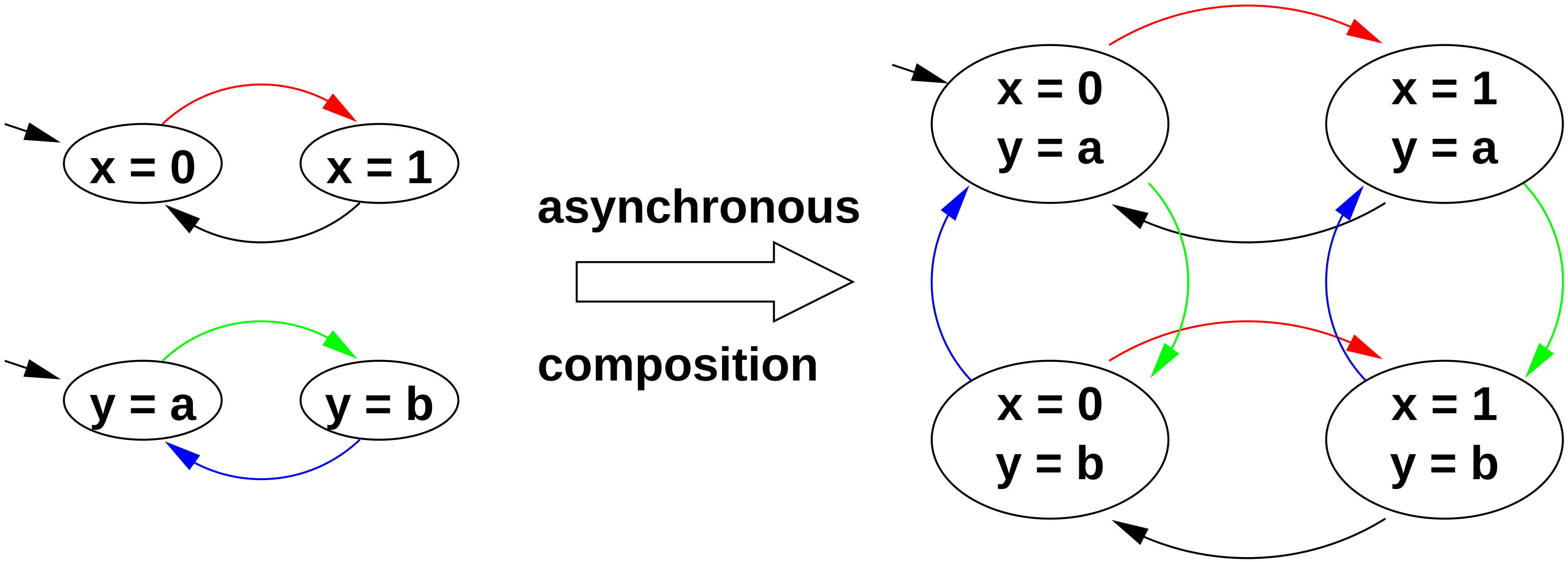
- Components evolve in parallel.
- At each time instant, every component performs a transition.



- Typical example: sequential hardware circuits.
- Synchronous composition is the default in NuSMV.

Asynchronous Composition

- Interleaving of evolution of components.
- At each time instant, one component is selected to perform a transition.

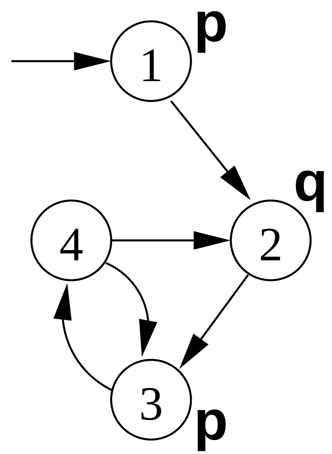


- Typical example: communication protocols.
- Asynchronous composition can be represented with NuSMV processes.

Model Checking

Model Checking is a formal verification technique where...

- ...the system is represented as Finite State Machine



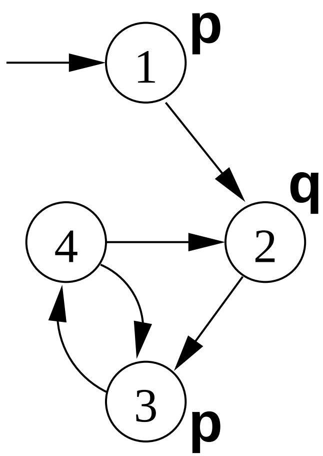
- ...the properties are expressed as temporal logic formulae

  LTL: **G(p −> Fq)** CTL: **AG(p −> AFq)**

- ...the model checking algorithm checks whether all the executions of the model satisfy the formula.

CTL Model Checking: Example

Consider a simple system and a specification:

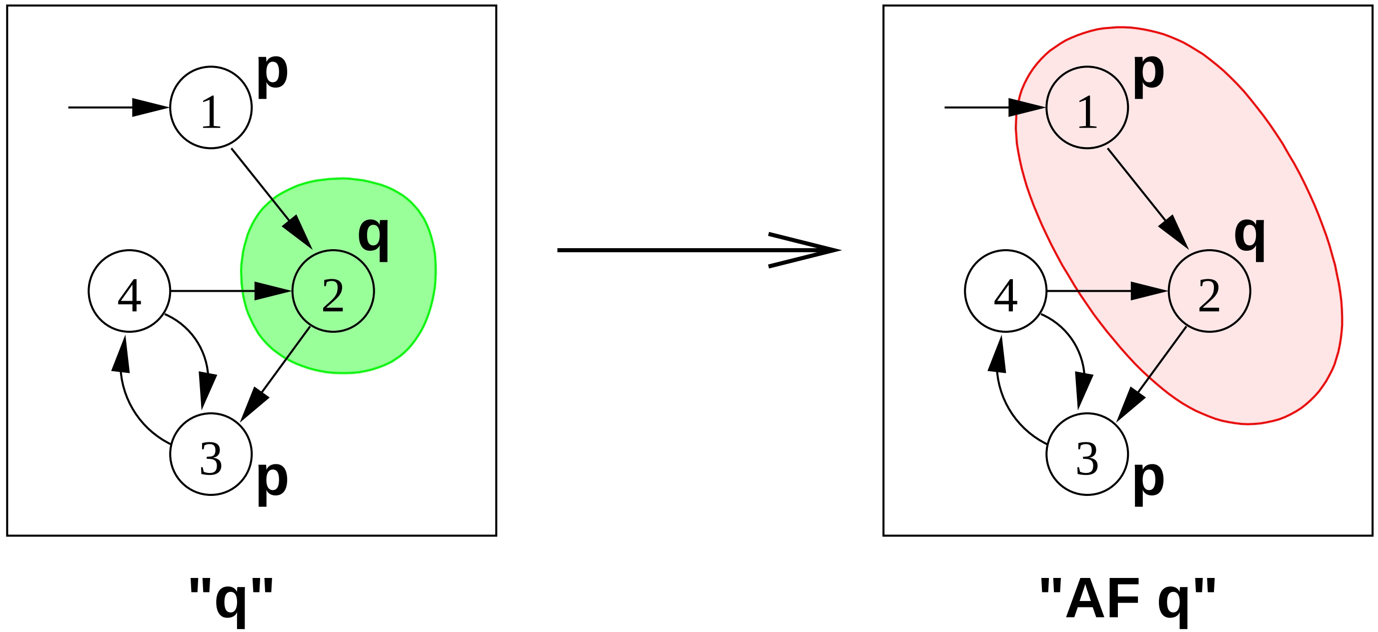


AG(p −> AFq)

Idea:

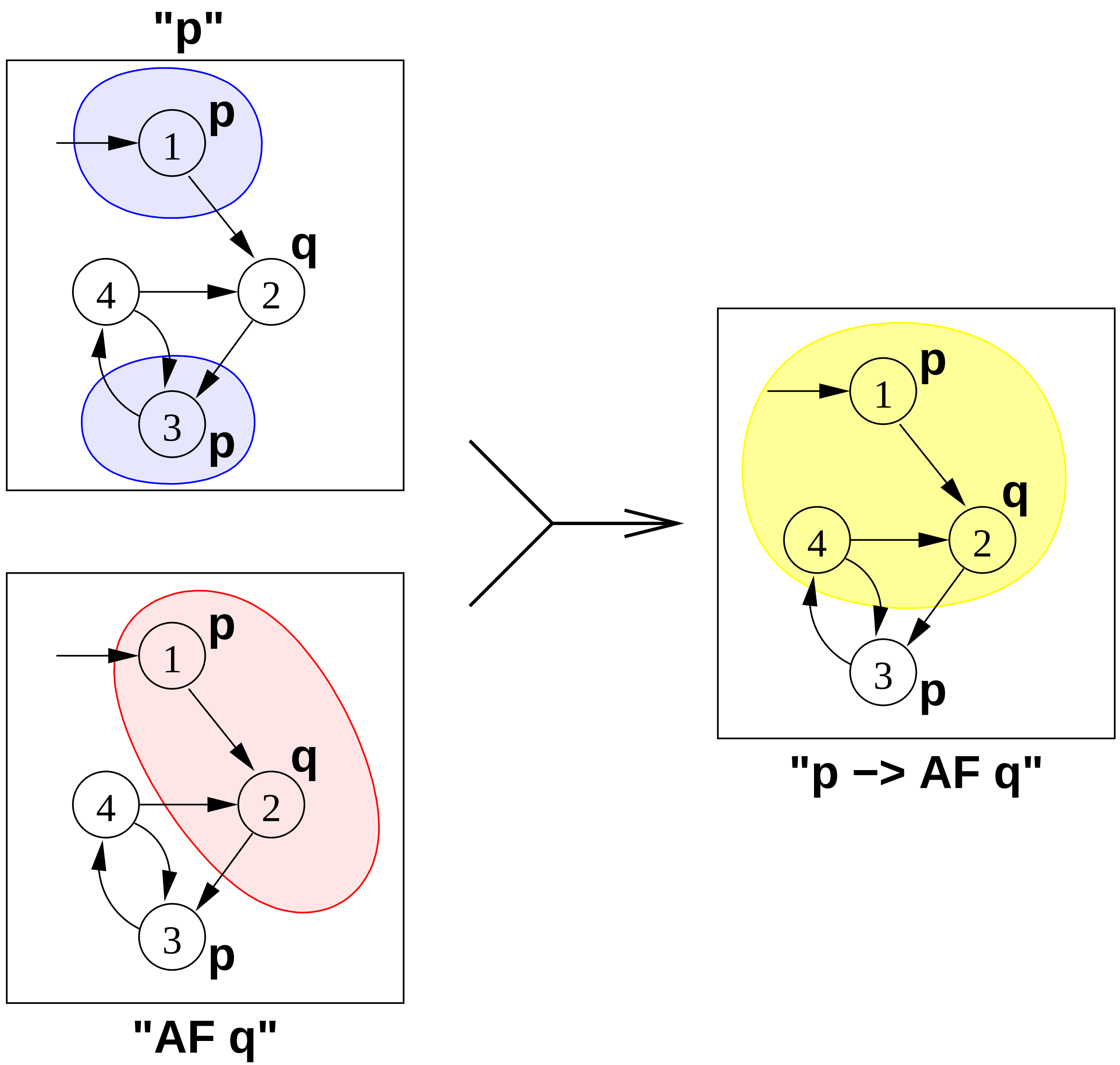
- construct the set of states where the formula holds
- proceeding “bottom-up” on the structure of the formula
- **q**, **AFq**, **p**, **p** $\rightarrow$ **AF q**, **AG(p** $\rightarrow$ **AF q)**

CTL Model Checking: Example

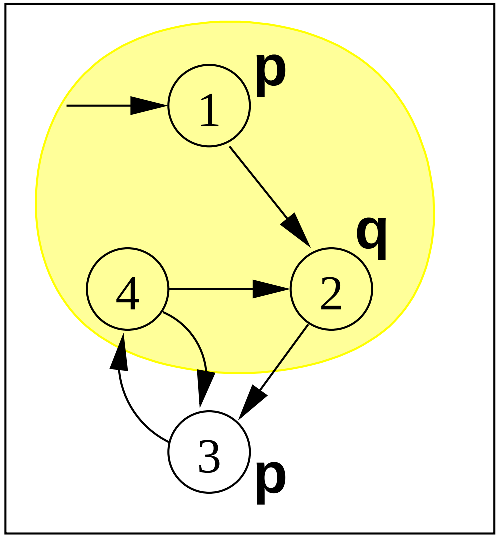


AF q is the union of **q**, **AX q**, **AX AX q**, ...

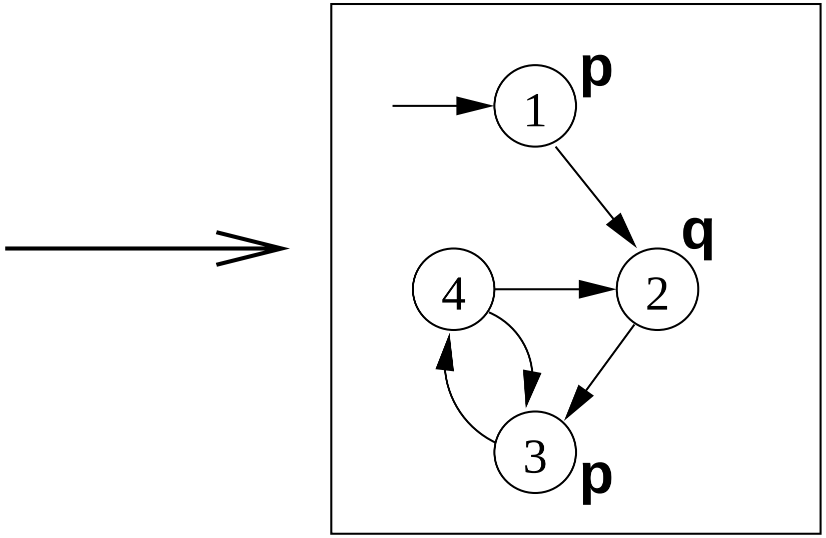
CTL Model Checking: Example



CTL Model Checking: Example



"p −> AF q"



"AG(p −> AF q)"

The set of states where the formula holds is empty!

Counterexample reconstruction is based on the intermediate sets.

Fix-Point Symbolic Model Checking

Model Checking Algorithm for CTL formulae based on fix-point computation:

- traverse formula structure, for each subformula build set of satisfying states; compare result with initial set of states.
- boolean connectives: apply corresponding boolean operation;
- on $\mathrm{AX}\,\Phi$, apply preimage computation
  - $\forall \mathbf{s}'.(\mathcal{T}(\mathbf{s}, \mathbf{s}') \rightarrow \Phi(\mathbf{s}'))$
- on $\mathrm{AF}\,\Phi$, compute least fixpoint using
  - $\mathrm{AF}\,\Phi \leftrightarrow (\Phi \vee \mathrm{AX}\,\mathrm{AF}\,\Phi)$
- on $\mathrm{AG}\,\Phi$, compute greatest fixpoint using
  - $\mathrm{AG}\,\Phi \leftrightarrow (\Phi \wedge \mathrm{AX}\,\mathrm{AG}\,\Phi)$

Part 3 - The NuSMV Model Checker

– Model Checking–

A Hands-On Introduction

A. Cimatti, M. Pistore, and M. Roveri

13^{th} International Conference on Automated Planning & Scheduling, June 10 2003, Trento (Italy)

Introduction

- NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.
- The NuSMV project aims at the development of a state-of-the-art model checker that:
  - is robust, open and customizable;
  - can be applied in technology transfer projects;
  - can be used as research tool in different domains.
- NuSMV is *OpenSource*:
  - developed by a distributed community,
  - “Free Software” license.

History: NuSMV 1

NuSMV is a reimplementation and extension of SMV.

- NuSMV started in 1998 as a joint project between ITC-IRST and CMU:
  - the starting point: SMV version 2.4.4.
  - SMV is the first BDD-based symbolic model checker (McMillan, 90).
- NuSMV version 1 has been released in July 1999.
  - limited to BDD-based model checking
  - extends and upgrades SMV along three dimensions:
    - functionalities (LTL, simulation)
    - architecture
    - implementation
- Results:
  - used for teaching courses and as basis for several PhD theses
  - interest by industrial companies and academics

History: NuSMV 2

- The NuSMV 2 project started in September 2000 with the following goals:
  - Introduction of SAT-based model checking
  - OpenSource licensing
  - Larger team (Univ. of Trento, Univ. of Genova, ...)
- NuSMV 2 has been released in November 2001.
  - first freely available model checker that combines BDD-based and SAT-based techniques
  - extended functionalities wrt NuSMV 1 (cone of influence, improved conjunctive partitioning, multiple FSM management)
- Results: in the first two months:
  - more than 60 new registrations of NuSMV users
  - more than 300 downloads

OpenSource License

The idea of OpenSource:

- The System is developed by a distributed community
- Notable examples: Netscape, Apache, Linux
- *Potential* benefits: shared development efforts, faster improvements...

Aim: provide a *publicly available*, *state-of-the-art* symbolic model checker.

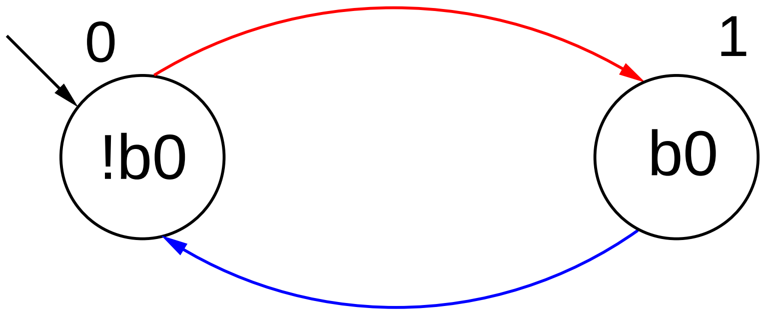
- *publicly available*: free usage in research and commercial applications
- *state of the art*: improvements should be made freely available

Distribution license for NuSMV 2: *GNU Lesser General Public License* (LGPL):

- anyone can freely download, copy, use, modify, and redistribute NuSMV 2
- any modification and extension should be made publicly available under the terms of LGPL ("copyleft")

The first SMV program

```
MODULE main
 VAR
   b0 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;
```



An SMV program consists of:

- Declarations of the state variables (`b0` in the example); the state variables determine the state space of the model.
- Assignments that define the valid initial states (`init(b0) := 0`).
- Assignments that define the transition relation (`next(b0) := !b0`).

Declaring state variables

The SMV language provides booleans, enumerative and bounded integers as data types:

boolean:

```
VAR
  x : boolean;
```

enumerative:

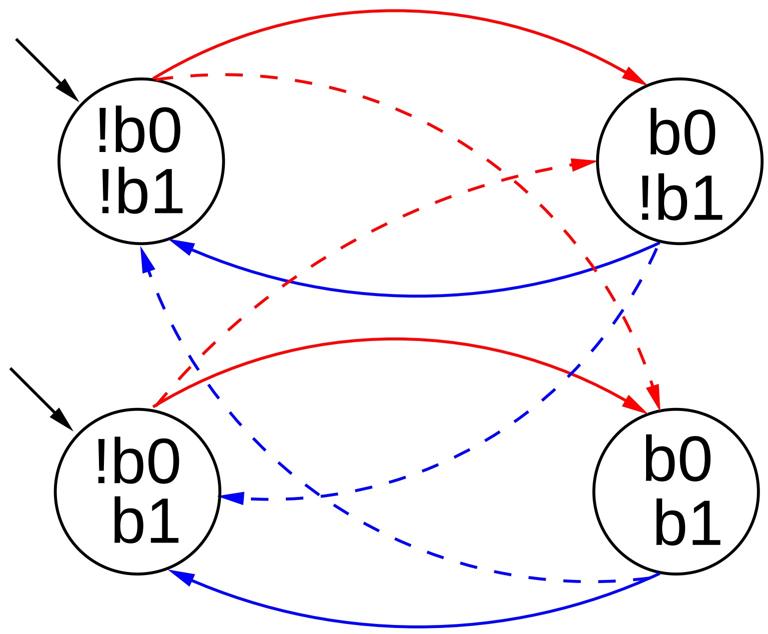
```
VAR
  st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

```
VAR
  n : 1..8;
```

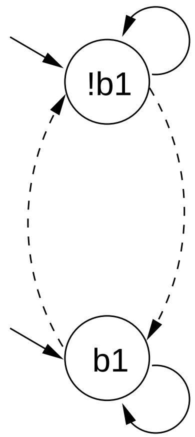
Adding a state variable

```
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;
```



Remarks:

- The new state space is the cartesian product of the ranges of the variables.
- Synchronous composition between the “subsystems” for b0 and b1.



Declaring the set of initial states

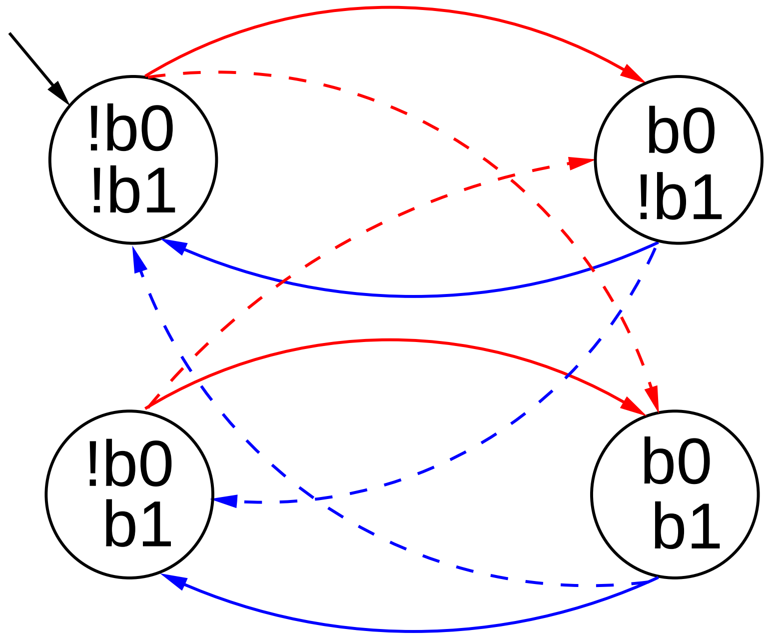
- For each variable, we constrain the values that it can assume in the *initial states*.

```
init(<variable>) := <simple_expression> ;
```

- <simple_expression> must evaluate to values in the domain of <variable>.
- If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.

Declaring the set of initial states

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
```



Expressions

- Arithmetic operators:

  `+` `-` `*` `/` `mod` `-` (unary)

- Comparison operators:

  `=` `!=` `>` `<` `<=` `>=`

- Logic operators:

  `&` `|` `xor` `!` (not) `->` `<->`

- Conditional expression:

```
case
  c1 : e1;
  c2 : e2;
  ...
  1  : en;
esac
```

  if `c1` then `e1` else if `c2` then `e2` else if `...` else `en`

- Set operators:

  `{v1,v2,...,vn}` (enumeration) `in` (set inclusion) `union` (set union)

Expressions

- Expressions in SMV do not necessarily evaluate to one value. In general, they can represent a set of possible values.

```
init(var) := {a,b,c} union {x,y,z} ;
```

- The meaning of `:=` in assignments is that the lhs can assume non-deterministically a value in the set of values represented by the rhs.
- A constant c is considered as a syntactic abbreviation for `{c}` (the singleton containing c).

Declaring the transition relation

- The transition relation is specified by constraining the values that variables can assume in the *next state*.

```
next(<variable>) := <next_expression> ;
```

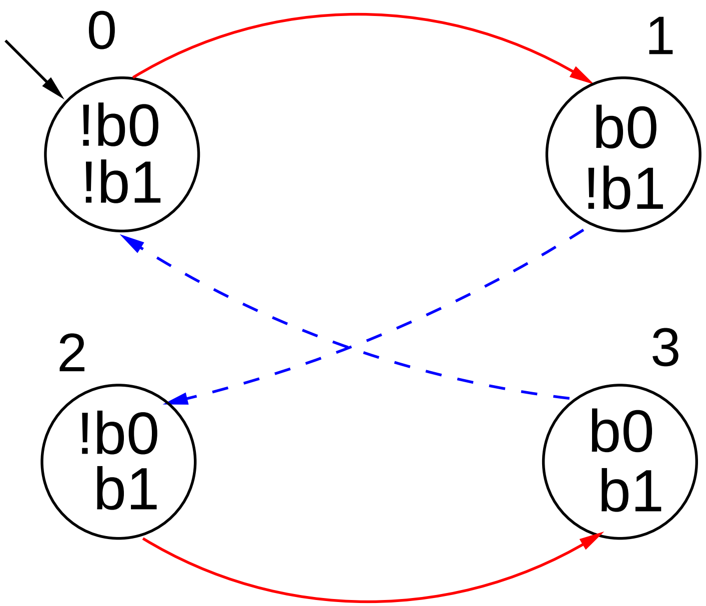
- <next_expression> must evaluate to values in the domain of <variable>.
- <next_expression> depends on “current” and “next” variables:

```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

- If no `next()` assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.
  Unconstrained variables can be used to model non-deterministic *inputs* to the system.

Declaring the transition relation

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
   next(b1) := ((!b0 & b1) | (b0 & !b1));
```



Specifying normal assignments

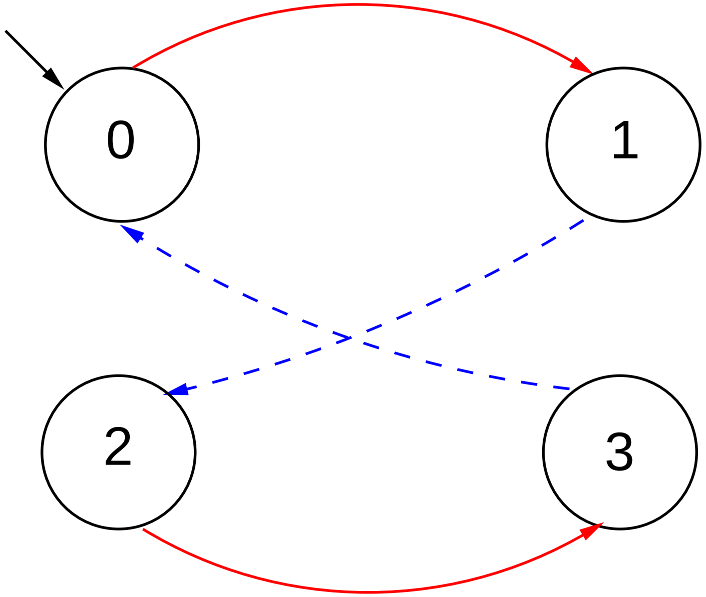
- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

```
<variable> := <simple_expression> ;
```

- `<simple_expression>` must evaluate to values in the domain of the `<variable>`.

Specifying normal assignments

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;
   out : 0..3;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
   next(b1) := ((!b0 & b1) | (b0 & !b1));

   out := b0 + 2*b1;
```



Restrictions on the ASSIGN

For technical reasons, the transition relation must be *total*, i.e., for every state there must be at least one successor state.

In order to guarantee that the transition relation is total, the following restrictions are applied to the SMV programs:

- ☞ Double assignments rule – Each variable may be assigned only once in the program.
- ☞ Circular dependencies rule – A variable cannot have “cycles” in its dependency graph that are not broken by delays.

If an SMV program does not respect these restrictions, an error is reported by NuSMV.

Double assignments rule

Each variable may be assigned only once in the program.

All of the following combinations of assignments are illegal:

```
init(status) := ready;
init(status) := busy;

next(status) := ready;
next(status) := busy;

status := ready;
status := busy;

init(status) := ready;
status := busy;

next(status) := ready;
status := busy;
```

Circular dependencies rule

A variable cannot have "cycles" in its dependency graph that are not broken by delays.

All the following combinations of assignments are illegal:

```
x := (x + 1) mod 2;

x := (y + 1) mod 2;
y := (x + 1) mod 2;

next(x) := x & next(x);

next(x) := x & next(y);
next(y) := y & next(x);
```

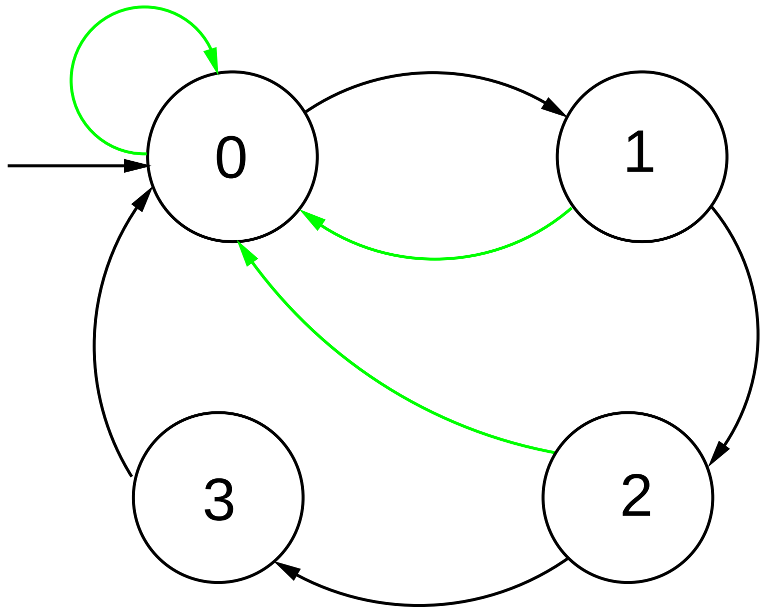
The following example is *legal*, instead:

```
next(x) := x & next(y);
next(y) := y & x;
```

The modulo 4 counter with reset

The counter can be reset by an external “uncontrollable” reset signal.

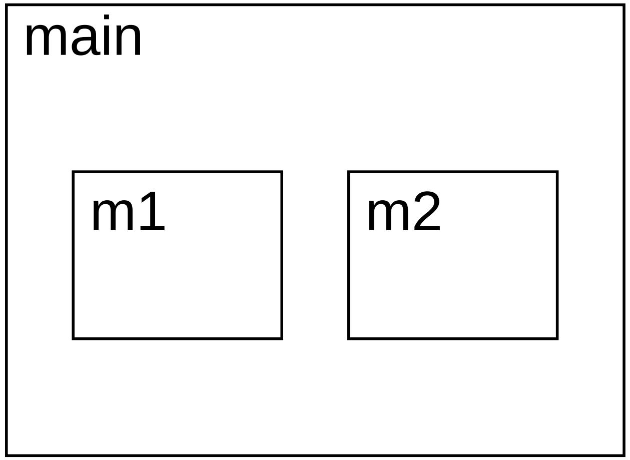
```
MODULE main
 VAR
   b0    : boolean;
   b1    : boolean;
   reset : boolean;
   out   : 0..3;

 ASSIGN
   init(b0) := 0;
   next(b0) := case
                 reset = 1 : 0;
                 reset = 0 : !b0;
               esac;

   init(b1) := 0;
   next(b1) := case
                 reset : 0;
                 1     : ((!b0 & b1) | (b0 & !b1));
               esac;

   out := b0 + 2*b1;
```



Modules

An SMV program can consist of one or more *module declarations*.

```
MODULE mod
  VAR out: 0..9;
  ASSIGN next(out) :=
               (out + 1) mod 10;
MODULE main
  VAR m1 : mod;
      m2 : mod;
      sum: 0..18;
  ASSIGN sum := m1.out + m2.out;
```

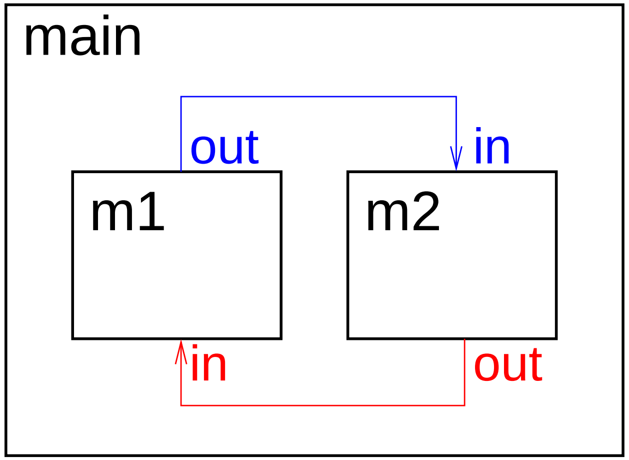


- Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- In each SMV specification there must be a module `main`. It is the top-most module.
- All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., `m1.out`, `m2.out`).

Module parameters

Module declarations may be *parametric*.

```
MODULE mod(in)
  VAR out: 0..9;
  ...
MODULE main
  VAR m1 : mod(m2.out);
      m2 : mod(m1.out);
  ...
```



- *Formal parameters* (`in`) are substituted with the *actual parameters* (`m2.out`, `m1.out`) when the module is instantiated.
- Actual parameters can be any legal expression.
- Actual parameters are passed by reference.

Example: The modulo 8 counter revisited

```
MODULE counter_cell(tick)

  VAR
    value : boolean;
    done  : boolean;

  ASSIGN
    init(value) := 0;
    next(value) := case
      tick = 0 : value;
      tick = 1 : (value + 1) mod 2;
    esac;

    done := tick & (((value + 1) mod 2) = 0);
```

Remarks:

☞ `tick` is the formal parameter of module `counter_cell`.

Example: The modulo 8 counter revisited

```
MODULE main
  VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;

  ASSIGN
    out := bit0.value + 2*bit1.value + 4*bit2.value;
```

Remarks:

- Module `counter_cell` is instantiated three times.
- In the instance `bit0`, the formal parameter tick is replaced with the actual parameter `1`.
- When a module is instantiated, all variables/symbols defined in it are preceded by the module instance name, so that they are unique to the instance.

Module hierarchies

A module can contain instances of others modules, that can contain instances of other modules... provided the module references are not circular.

```
MODULE counter_8 (tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out  := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

MODULE counter_512(tick) -- A counter modulo 512
  VAR
    b0 : counter_8(tick);
    b1 : counter_8(b0.done);
    b2 : counter_8(b1.done);
    out : 0..511;
 ASSIGN
    out := b0.out + 8*b1.out + 64*b2.out;
```

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`)

Invariant specifications

- Invariant properties are specified via the keyword `INVARSPEC`:

```
INVARSPEC <simple_expression>
```

- Example:

```
MODULE counter_cell(tick)
  ...
MODULE counter_8(tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out  := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

  INVARSPEC
    done <-> (bit0.done & bit1.done & bit2.done)
```

LTL specifications

- LTL properties are specified via the keyword LTLSPEC:

```
LTLSPEC <ltl_expression>
```

where <ltl_expression> can contain the following temporal operators:

```
X _   F _   G _   _ U _
```

- 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 properties

☞ CTL properties are specified via the keyword SPEC:

```
SPEC <ctl_expression>
```

where <ctl_expression> can contain the following temporal operators:

```
AX _   AF _   AG _   A[_ U _]
EX _   EF _   EG _   E[_ U _]
```

☞ 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
```

Fairness Constraints

Let us consider again the counter with reset.

- ☞ The specification AF `out = 1` is not verified.
- ☞ On the path where `reset` is always `1`, then the system loops on a state where `out = 0`, since the counter is always reset:

  `reset = 1,1,1,1,1,1,1...`

  `out = 0,0,0,0,0,0,0...`
- ☞ Similar considerations hold for the property AF `out = 2`. For instance, the sequence:

  `reset = 0,1,0,1,0,1,0...`

  generates the loop:

  `out = 0,1,0,1,0,1,0...`

  which is a counterexample to the given formula.

Fairness Constraints

- NuSMV allows to specify *fairness* constraints.
- Fairness constraints are formulas which are assumed to be true infinitely often in all the execution paths of interest.
- During the verification of properties, NuSMV considers path quantifiers to apply only to fair paths.
- Fairness constraints are specified as follows:

```
FAIRNESS <simple_expression>
```

Fairness Constraints

- With the fairness constraint

```
FAIRNESS
  out = 1
```

  we restrict our analysis to paths in which the property `out = 1` is true infinitely often.
- The property `AF out = 1` under this fairness constraint is now verified.
- The property `AF out = 2` is still not verified.
- Adding the fairness constraint `out = 2`, then also the property `AF out = 2` is verified.

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 boolean;
  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 ASSIGNs.

Assignments versus constraints

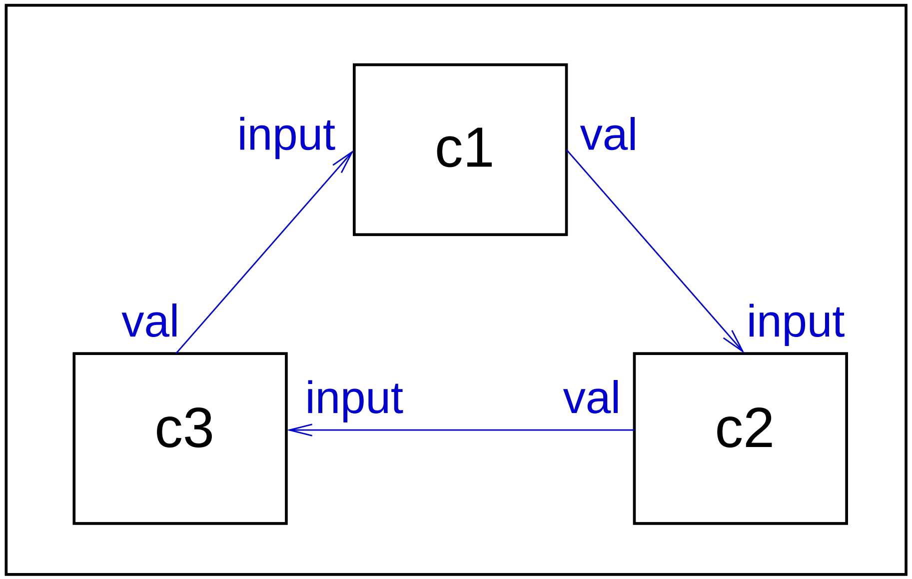
- Models written in **assignment style**:
  - by construction, there is always *at least one initial state*;
  - by construction, all states have *at least one next state*;
  - *non-determinism is apparent* (unassigned variables, set assignments...).
- Models written in **constraint style**:
  - `INIT` 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	*running*	*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

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/`