MCT99

A Hands-on Tutorial on Model Checking
— NuSMV —

FLoC'99

2-4 July, Trento, Italy

Alessandro Cimatti

Marco Pistore

Marco Roveri

Part I: Basic of NuSMV and SMV language

An Overview of NUSMV

- NUSMV is an OBDD-based symbolic model checker.
- Uses a structured language to describe finite-state systems.
- It is oriented to the verification of synchronous systems.
- Allows to model check CTL and LTL specifications.
- When the specification is not satisfied, it produces a counterexample.
- Allows to simulate the specified model.

An Overview of NUSMV

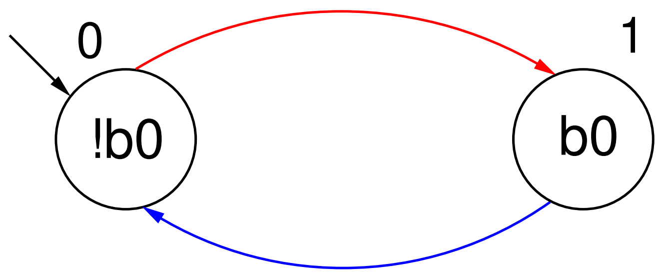
- NUSMV is a reengineering/reimplementation and extension of SMV [McM93].
- SMV was developed by K. McMillan at CMU. It was the first symbolic model checker.
- NUSMV is the result of a joint project between CMU and ITC-IRST involving A. Cimatti, E. Clarke, F. Giunchiglia, A. Morichetti, M. Roveri.
- NUSMV is under ongoing development.
- NUSMV is available at the URL:

`http://afrodite.itc.it:1024/~nusmv`

The first SMV program

```
MODULE main
 VAR
   b0 : boolean;

 ASSIGN
   init(b0) := 0;
   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};
```

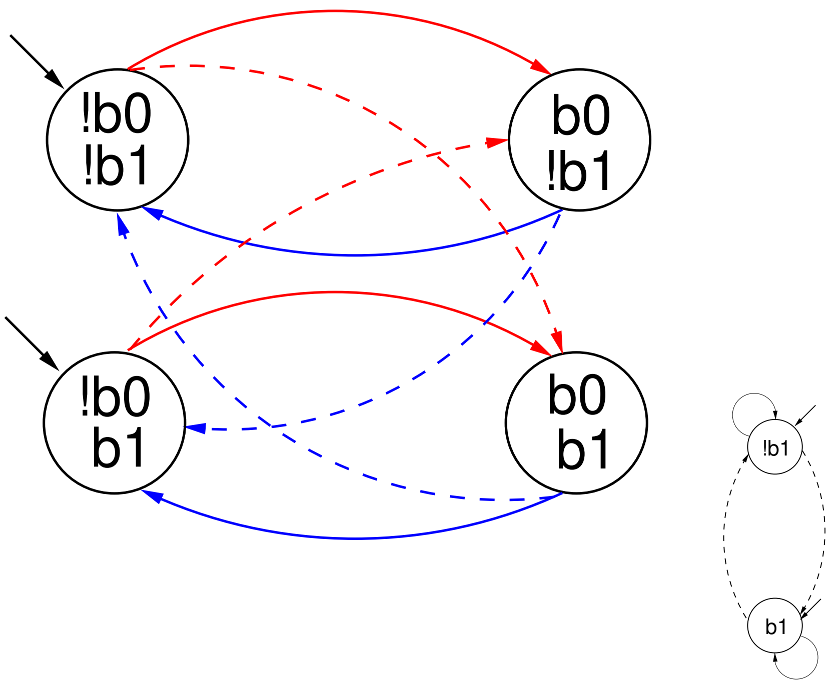
integers (bounded):

```
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> ;
```

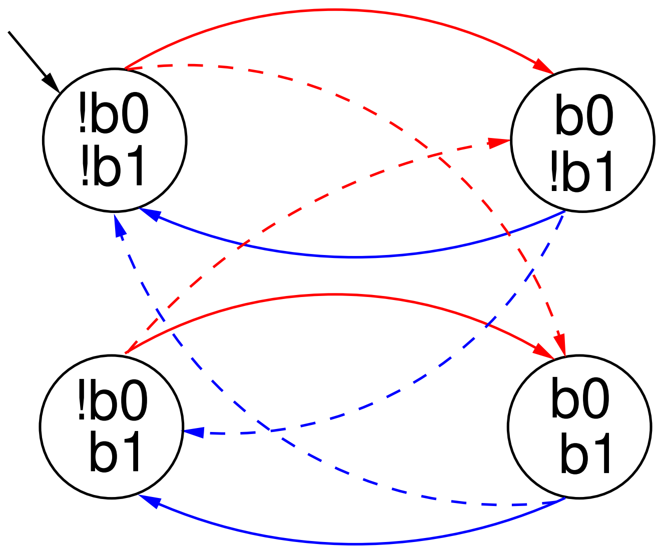
- `<simple_expression>` must evaluate to values in the domain of `<variable>`.
- If the initial values for a variable are not specified, then they can be any in the domain of the variable.

Declaring the set of initial states

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

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

   init(b1) := 0;
```



Expressions

- Expressions in SMV do not necessarily evaluate to one value.
- They can represent several possible values.
- Alternatively, we can say that an expression evaluates to a set of values, from which one may be choosen.
- A constant `c` is considered as a synctatic abbreviation for `{c}` (the singleton containing `c`).
- The meaning of `:=` in assignements is that the lhs can assume non deterministically a value in the set the rhs evaluate to.

```
init(x) := {1,2,3};
```

Expression (cont.)

- Arithmetic operators:

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

- Comparison operators:

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

- Logic operators:

  `&` `|` `!` (unary, logical not) `->` `<->`

- Set operators:

  `in` (set inclusion) `union` (set union)

- Conditional expression:

```
case
  e1 : e2;
  e3 : e4;
  ...
esac
```

if `e1` then `e2` else if `e3` then `e4` else ...

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> ;
```

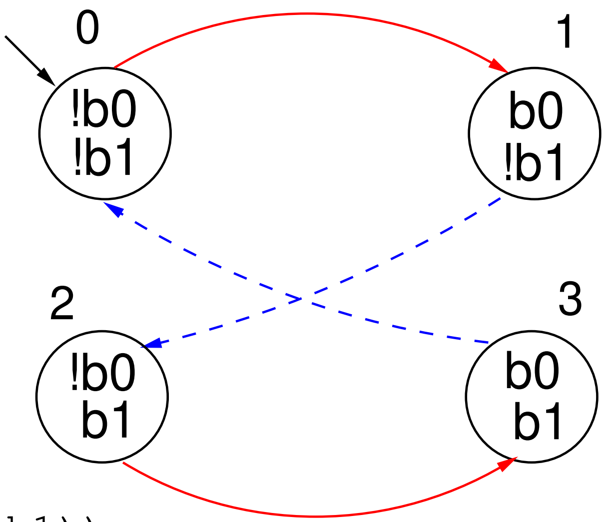
- `<next_expression>` depends on "current" and "next" variables;
- it must evaluate to values in the domain of `<variable>`.
- If no `next()` assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.
- Unconstrained variables can be used to 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 assignements

- Normal assignements constrain the current value of a variable to the value of other current variables.
- They can be used to model outputs of the form $O(\underline{y}, \underline{x})$, where $\underline{y}$ is the vector of output variables, and $\underline{x}$ is the vector of state variables.
- Normal assignements have the following form:

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

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

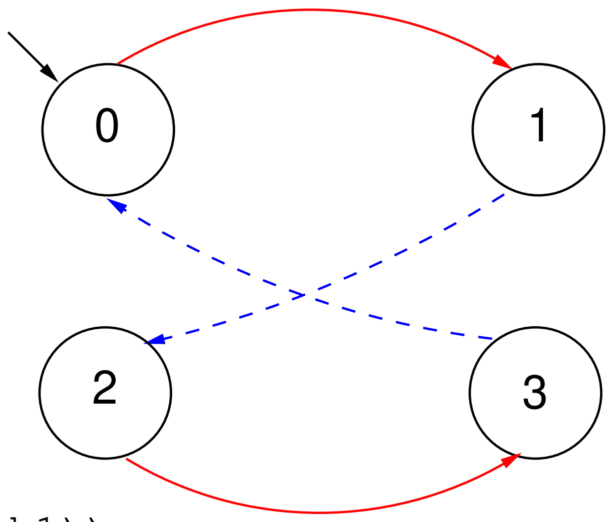
Specifying normal assignements

```
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.

To guarantee that the specified program yields a total transition relation, the following restrictions are applied:

- Single assignment 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.

Single assignment rule

None of the following combinations of assignments is legal:

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

Circular dependencies rule

None of the following combinations of assignments is legal:

`x := (x + 1) mod 2;`	`next(x) := z & next(x);`

`x := (y + 1) mod 3;`
`y := (x - 1) mod 3;`
	`next(x) := x & next(y);`
	`next(y) := next(z) & y;`
	`next(z) := z & next(x);`

The modulo 4 counter with reset

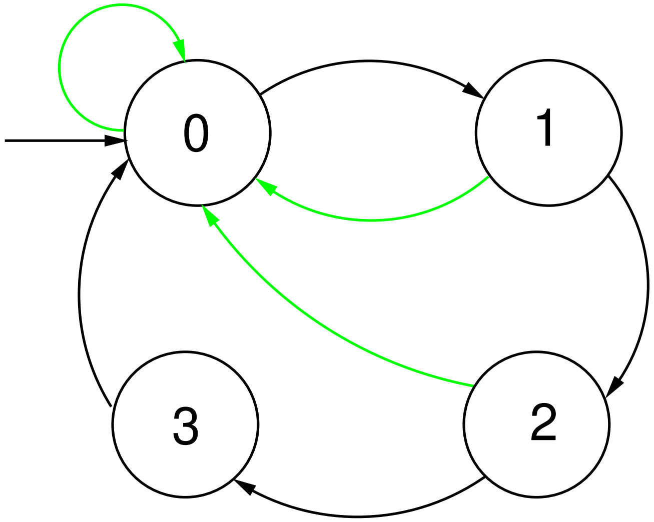
The counter can be reset by an external reset signal.

```
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;
```



Specifying CTL properties

- ☞ 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 reachable.

```
SPEC
  AF out = 3
```

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

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

- ☞ An execution leading to a state in which `out = 3` can be generated with the specification:

```
SPEC
  !EF out = 3
```

CTL specifications

NUSMV provides bounded 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 reachable in 2 steps.

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

- ☞ Invariantly a state in which `out = 3` is reachable in 3 steps.

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

Quantitative characteristics computations

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

Introducing 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...
```

☞ Analogous 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 CTL 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 <ctl_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.

Exercise 1: Mutual Exclusion

A two-users mutual exclusion:

- We have two users `U1` and `U2`, and an Arbiter.
- Each user can be either `NonCritical`, `Trying` or `Critical`.
- From `NonCritical`, it can nondeterministically go to `Trying`.
- From `Trying`, it can go to `Critical` when enabled by the arbiter.
- From `Critical`, it goes back to `NonCritical` in at most 4 time units.

Design the Arbiter, the users, and . . .

- Verify that the two users cannot be at the same time in their `Critical` section.
- Verify that if a user tries to enter its critical section, it will eventually succeed.
- Compute the maximum time needed to enter the critical section.

Part II: Advanced features of NuSMV

The `DEFINE` declaration

- ☞ Defined symbols do not require extra BDD variables creation.
- ☞ Each occurrence of a defined symbol in a specification is replaced with the body of the definition.
- ☞ The BDD corresponding to the body of defined symbols becames part of each expression using the defined symbol.
- ☞ It is similar to a macro definition.

The DEFINE declaration

```
MODULE main
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;
  reset := b0 & b1 & b2;
```

Arrays

The SMV language provides also the possibility to define arrays of basic data types (and arrays of 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(z[3][2]) := {gree, orange};
```

☞ Remark: Array indexes in SMV must be constants.

Using propositional formulas to specify M

The SMV language allows to specify M using propositional formulas:

```
<init_declaration>      ::- 'INIT' <simple_expr>
<trans_declaration>     ::- 'TRANS' <next_expr>
<invar_declaration>     ::- 'INVAR' <simple_expr>
```

- ☞ No checks performed on these formulas.
- ☞ Logical absurdities can be introduced.
- ☞ Not recommended unless you know what you are doing.
- ☞ Very useful for writing translators from other languages to NuSMV.

Using propositional formulas to specify M

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;
```

is equivalent to:

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

Modules

- Modules in SMV are a bundle of definitions that can be reused.
- Modules are composed by instantiating them inside another module.
- Instantiation is performed inside the `VAR` declaration of a module.
- They can have *formal parameters*, which are substituted with the actual parameters when the module is instantiated.
- Actual parameters can be any legal expression.
- The semantic of module instantiation and parameters passing is similar to call-by-reference.
- All the variables of an instance of a module can be passed to another module by passing the module instance name as argument.

Example: The modulo 8 counter revisited

```
MODULE counter_cell(tick)

VAR
  value : boolean;

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

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

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

Example: The modulo 8 counter revisited

Module `counter_cell` is instantiated three times.

```
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;
```

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 preceeded by the module instance name, so that they are unique to the instance.

Restriction on `MODULE` declaration

- There must be a module `main` in a SMV specification.
- The module `main` has a special meaning in the SMV language, in the same way that it does in the `C` programming language.
- There must be no recursively defined modules.
- You cannot instantiate with the same name two modules.
- All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation.

Module hierarchies

A module can contain instances of others 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;

DEFINE
  done := bit2.done;

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

Module hierarchies (cont.)

```
-- A counter modulo 512

MODULE main
  VAR
    b0 : counter_8(1);
    b1 : counter_8(b0.done);
    b2 : counter_8(b1.done);
    out : 0..511;

ASSIGN
  out := b0.out + 8*b1.out + 64*b2.out;

SPEC
  AF b2.bit2.done

SPEC
  !EF out = 511
```

Exercise 2

Use modules and arrays to

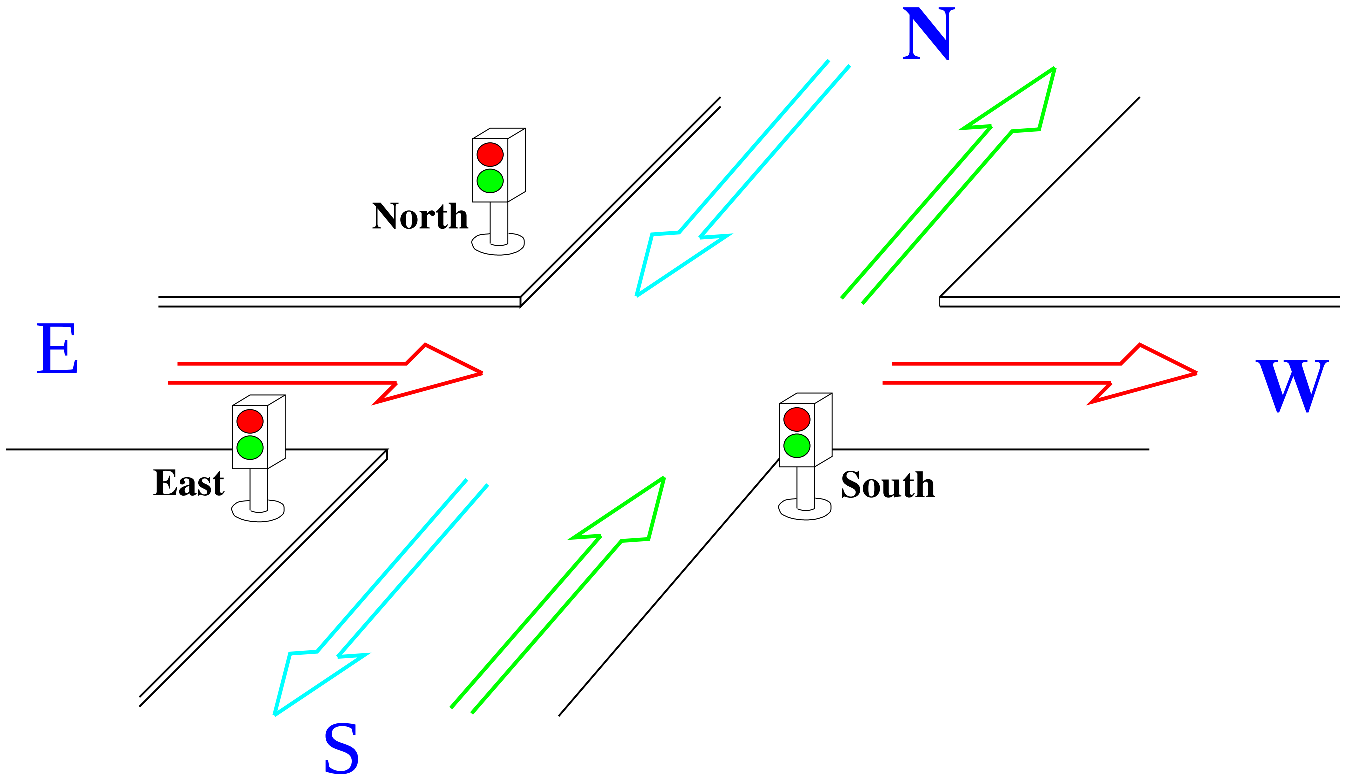
- Extend the mutual exclusion controller of exercise 1 to a generic number of users.
- Implement the read-write lock algorithm described below:

  A Readers/Writer lock controls the access to a resource (for instance a cache memory) that can be read simultaneously by many users, but that can be written only by a user that has an exclusive access.

  In any moment the resource can be:
  - either unused,
  - or accessed in read mode by one or more users,
  - or accessed in read-write mode by an exclusive user.

Exercise 3

Design a traffic lights controller of an intersection where a two-way street running north and south intersects a one way street running West, such that collisions are avoided and cars do not wait forever at red light.



Exercise 3 – hints

- ☞ We need a sensor for each direction to indicate the presence of a car in that direction. It can be considered an input to the system.
- ☞ We need for each direction a variable to indicate the color of the light.
- ☞ We need a lock that is set when the traffic is enabled in North-South directions, and prevents East light to became green.
- ☞ We need a boolean variable for each direction to remember if there has been or not a request.

Partitioning the transition relation

The basic computation in symbolic model checking is the relational product:

$$S(\underline{x}') = \exists \underline{x}.(S(\underline{x}) \wedge R(\underline{x}, \underline{x}'))$$

- ☞ There are efficient algorithms for computing $S(\underline{x}')$, without building the conjunction, but . . .
  - operation is complex;
  - monolithic R sometimes is too large;
- ☞ Avoid building R by partitioning.

Conjunctive Partitioning

Represent R as an implicitly conjoined list of smaller transition relation.

$$R(\underline{x}, \underline{x}') = \bigwedge_{i=0}^{n} R_i(\underline{x}, \underline{x}')$$

and thus perform early quantification:

$$\begin{aligned}
S(\underline{x}') &= \exists \underline{x}.(S(\underline{x}) \wedge R(\underline{x}, \underline{x}')) &= \\
&= \exists \underline{x}.(S(\underline{x}) \wedge \textstyle\bigwedge_{i=0}^{n} R_i(\underline{x}, \underline{x}')) &= \\
&= \underbrace{\exists \underline{x}_n(\ldots \underbrace{\exists \underline{x}_1 \underbrace{(\exists \underline{x}_0(S(\underline{x}) \wedge R_0(\underline{x}, \underline{x}'))}_{S_0} \wedge R_1(\underline{x}, \underline{x}'))}_{S_1} \ldots \wedge R_n(\underline{x}, \underline{x}'))}_{S_n}
\end{aligned}$$

Conjunctive partitioning in NuSMV

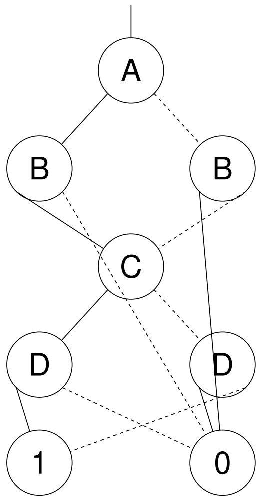
NuSMV provides two conjunctive partioning heuristics.

- Conjunctive partitioning in NuSMV is actually only available for synchronous systems.
- Individual **next()** assignments are grouped into partitions whose size in terms of BDD nodes does not exceed a given threshold.
- A refinement of the above. Individual **next()** assignments are grouped into partitions whose size in terms of BDD nodes does not exceed a given threshold. Then the partitions are ordered according to an heuristic in order to improve image computation efficiency [RAP+95].

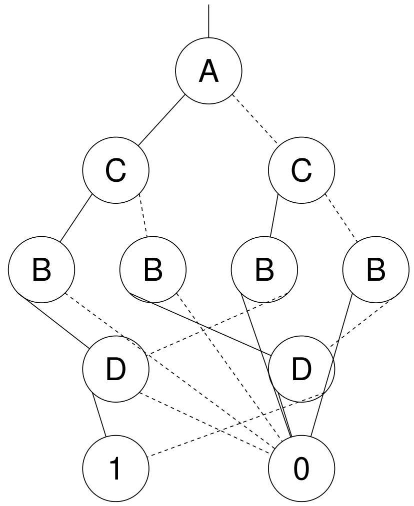
Variable Ordering

- NuSMV uses BDDs to encode states.
- The order of the varibles can have a big inpact on the size of the BDD (for a given formula) and thus on the performances.

$$(\mathtt{A} \leftrightarrow \mathtt{B}) \land (\mathtt{C} \leftrightarrow \mathtt{D})$$



A, B, C, D



A, C, B, D

Variable Ordering

The default ordering used by NuSMV is the order in which the variables are declared in a depth-first traversal of the module hierarchy starting from the module `main`.

```
MODULE counter_cell(tick)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := case
    tick = 0 : value;
    tick = 1 : (value + 1) mod 2;
  esac;
DEFINE
  done := tick & (((value + 1) mod 2) = 0);

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;
```

The default ordering is: `bit0.value`, `bit1.value`, `bit2.value`, `out`

Variables Ordering

- NUSMV allows the user to specify the order of the variables.
- A "good" variable order can be specified by hand or generated automatically by automatic variable reordering algorithms (NUSMV provides 19 different algorithms).
- If in the previous example we use the variable ordering:

  `bit0.value`, `out`, `bit1.value`, `bit2.value`

  The the size of the transition relation changes from 32 BDD nodes to 23 BDD nodes.