Reference material:

- Tutorial material from the nuSMV web site http://nusmv.fbk.eu/NuSMV/tutorial/index.html
- Notes of prof. Alessandro Artale in Bozen: http://www.inf.unibz.it/~artale/FM/fm.htm

Verifica dei Programmi Concorrenti 19-20

NuSMV

Prof.ssa Susanna Donatelli
Universita' di Torino
www.di.unito.it
susi@di.unito.it

nuSMV

NUSMV is a symbolic model checker developed as a joint project between the Formal Methods group at ITC-IRST, the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at University of Geneva and the Mechanized Reasoning Group at University of Trento.

Current version: 2.6.0

Tutorial 2.6

The input language are state machines

From the website:

NuSMV is a symbolic model checker developed as a joint project between:

- the Embedded Systems Unit in the Center for Information Technology at FBK-IRST
- the Model Checking group at Carnegie Mellon University ,
- the Mechanized Reasoning Group at University of Genova
- The Mechanized Reasoning Group at University of Trento.

NuSMV is a reimplementation and extension of SMV, the first model checker based on BDDs. NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas.

.

From the website:

NuSMV2, combines BDD-based model checking component that exploits the CUDD library developed by Fabio Somenzi at Colorado University and SAT-based model checking component that includes an RBC-based Bounded Model Checker, which can be connected to the Minisat SAT Solver and/or to the ZChaff SAT Solver. The University of Genova has contributed SIM, a state-of-the-art SAT solver used until version 2.5.0, and the RBC package use in the Bounded Model Checking algorithms.

nuSMV – input language

State machines are defined by a guarded-command language. NUSMVconsists of one or more modules and one must be called **main**.

An SMV program consists of:

- Type declarations of the system variables;
- Assignments that define the valid initial states (e.g., `init(b0) := 0`).
- Assignments that define the transition relation (e.g., `next(b0) := !b0`).
- They can be **Non-Deterministic**: Several values in braces.
- CTL or LTL specifications introduced by the keywords SPEC, LTLSPEC, respectively.

nuSMV

- NUSMV takes the specification of a model and a set of properties (either in CTL or LTL) as input.
- NUSMV output either *True* if the properties hold or *False* with a trace showing the failure.
- The set of states correspond to the set of all possible values for the variables.
- NUSMV uses !,&, |, -> for the boolean not, and, or, implies.
- NUSMV uses **G,F,X,U,A,E** as defined before

nuSMV

- NUSMV breaks a system description into *modules*.
- A module is instantiated when a variable having the module as its type is declared.
- Modules can have parameters.
- The notation `module-name.x` is used to access the variable `x` of the `module-name`.
- The keyword DEFINE is used to assign (the current value of) an expression to a symbol without the need to introduce a variable.
- Defined symbols refer just to an expression then they cannot be assigned non-deterministically.

nuSMV

- Modules, by default, are composed *synchronously*
- Each of the modules execute in parallel (e.g., the counter example).
- Using the keyword *`process`* modules are composed asynchronously (interleaving semantics): at each tick one of them is non-deterministically chosen and executed.

- The main use of NUSMV is through an *interactive* shell.
- The user has the possibility to:
  - Explore the possible executions called *Traces*;
  - Construct the Model;
  - Check specification and/or build counterexamples;

nuSMV: first example

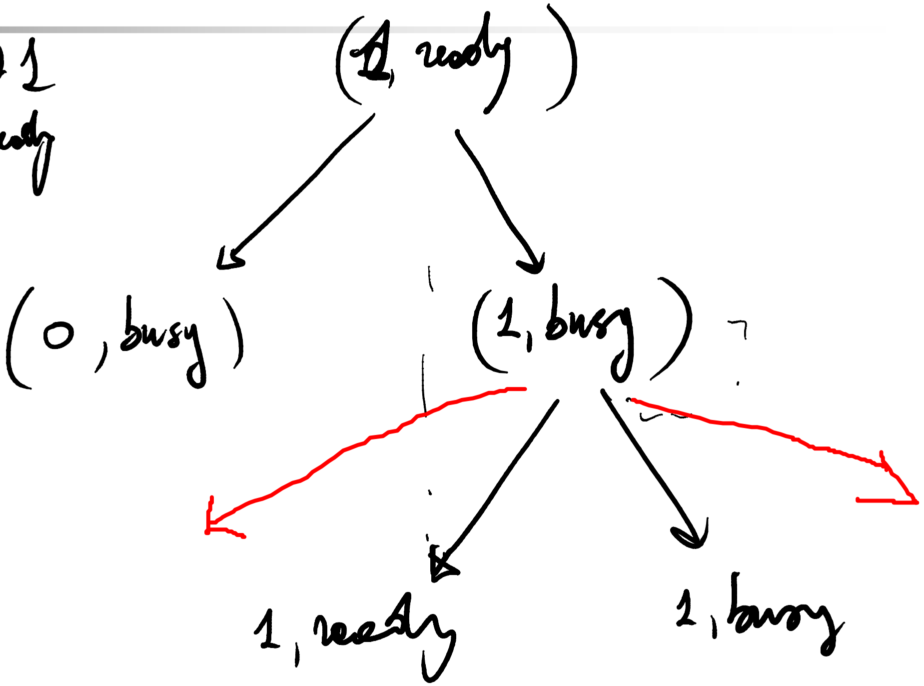
```
MODULE main
VAR
    request : boolean;
    state :  {ready,busy};
ASSIGN
    init(state) := ready;
    next(state):= case
                          state=ready & request=1: busy;
                          TRUE                   : {ready, busy}
                 esac;
```

The variable request is not assigned. This means that there are no constraints on its values, and thus it can assume any value. request is thus an unconstrained input to the system.

nuSMV: first example

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

rep = ~~0~~ 1
state = ready



```
MODULE main
VAR
  request : {Tr, Fa};
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request = Tr): busy;
                   TRUE : {ready,busy};
                 esac;
SPEC
  AG((request = Tr) -> AF state = busy
```

nuSMV: second example

```
MODULE counter_cell (carry_in)
     VAR
        value : boolean;
     ASSIGN
        init(value) := 0;
        next(value):= (value + carry_in) mod 2;
     DEFINE
        carry_out := value & carry_in;
MODULE  main
     VAR
        bit0 : counter_cell(1);
        bit1 : counter_cell(bit0.carry_out);
        bit2 : counter_cell(bit1.carry_out);
```

```
MODULE counter_cell (carry_in)
        VAR
             value : boolean;
        ASSIGN
             init(value) := 0;
             next(value):= (value + carry_in) mod 2;
        DEFINE
             carry_out := value & carry_in;
MODULE  main
        VAR bit0 : counter_cell(1);
               bit1 : counter_cell(bit0.carry_out);
               bit2 : counter_cell(bit1.carry_out);
```

bit0.carry_in = 1
bit1.carry_in = bit0.value
bit2.carry_in = bit1.value & bit1.carry_in

bit0.carry_out = bit0.value
bit1.carry_out = bit1.value & bit1.carry_in
bit2.carry_out = bit2.value & bit2.carry_in

	value	**Carry_in**	**Carry_out**
bit0	0	1	0
bit1	0	0	0
bit2	0	0	0
bit0	1	1	0
bit1			
bit2			
bit0			
bit1			
bit2			
bit0			
bit1			
bit2			
bit0			
bit1			
bit2			

nuSMV: asynchronous example

nuSMV allows for synchronous behaviour (as in previous example) as well as asynchronous, through the keyword "process"

It is implicit that if a given variable is not assigned by the process in a step, , then its value remains unchanged.

nuSMV: asynchronous example

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
   init(output) := 0;
   next(output) := !input;
MODULE main
 VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);
```

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
   init(output) := 0;
   next(output) := !input;
MODULE main
 VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);
```

gate1.output = !gate1.input = !gate3.output
gate2.output = !gate2.input = !gate1.output
gate3.output = !gate3.input = !gate2.output

	output	**input**	
gate1	0	0	
gate2	0	0	
gate3	0	0	
gate1	0	0	
gate2	0	0	
gate3	1	0	
gate1	0	1	
gate2	1	0	
gate3	1	0	
gate1		1	
gate2		0	
gate3		1	
gate1			
gate2			
gate3			

nuSMV: fairness - justice

In asynchronous nuSMV a process is not "forced" to move (and therefore also unrealistic sequences are considered)

A fairness contraints can be specified:

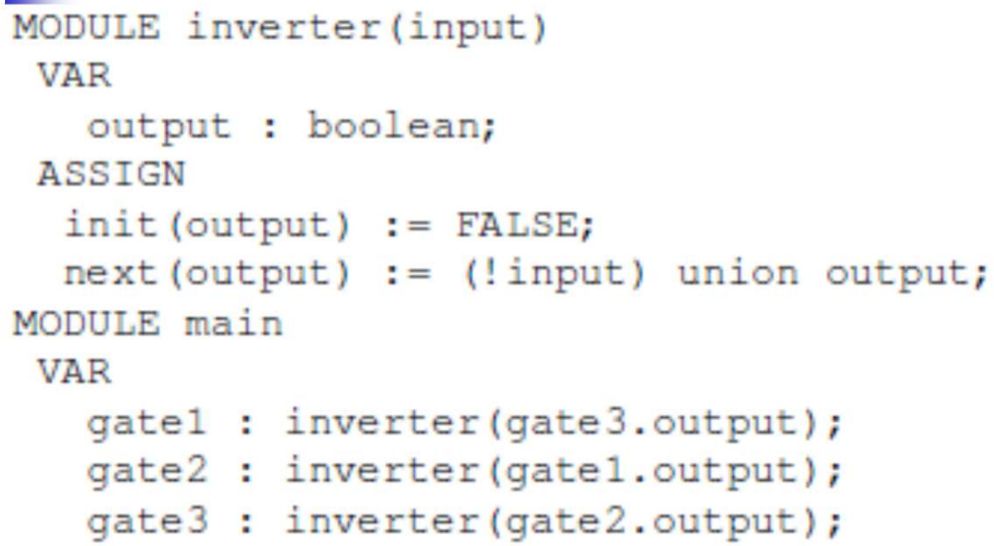
FAIRNESS f

Implies that only paths in which formula f is infinitely often true are considered by the model-checker

Example: FAIRNESS running

Where "running" is a predefined variable associated to each process, with the obvious meaning

Synchronous inverter and fairness



```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
  init(output) := FALSE;
  next(output) := (!input) union output;
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate1.output);
   gate3 : inverter(gate2.output);
```

In this case we cannot force the inverters to be effectively active infinitely often using a fairness declaration. In fact, a valid scenario for the synchronous model is the one where all the inverters are idle and assign to the next output the current value of output.

nuSMV: fairness - compassion

In NUSMV we can also specify compassion

COMPASSION p q

Implies that only paths in which GF p ➔ GF q is satisfied are considered by the model-checker

Example: COMPASSION ask receive

means that only paths that satisfy

GF ask ➔ GF receive

are considered for model checking.

NOTE: not available for CTL model checking (guess why…)

nuSMV: semaphore

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
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;
       1                             : state;
     esac;
   next(semaphore) :=
     case
       state = entering : 1;
       state = exiting  : 0;
       1                : semaphore;
     esac;
 FAIRNESS
   running
```

nuSMV: traces

Three simulation modes (how to select a state):

- random
- deterministic,
- interactive

In deterministic simulation mode the first state of a set (whatever it is) is chosen, while in the random one the choice is performed nondeterministically. Traces are automatically generated by NUSMV: the user obtains the whole of the trace in a time without control over the generation itself (except for the simulation mode and the number of states entered via command line).

In the third simulation mode, the user has a complete control over the trace, as it can choose the next step of the execution

nuSMV: traces

For the very first example (called short.smv)

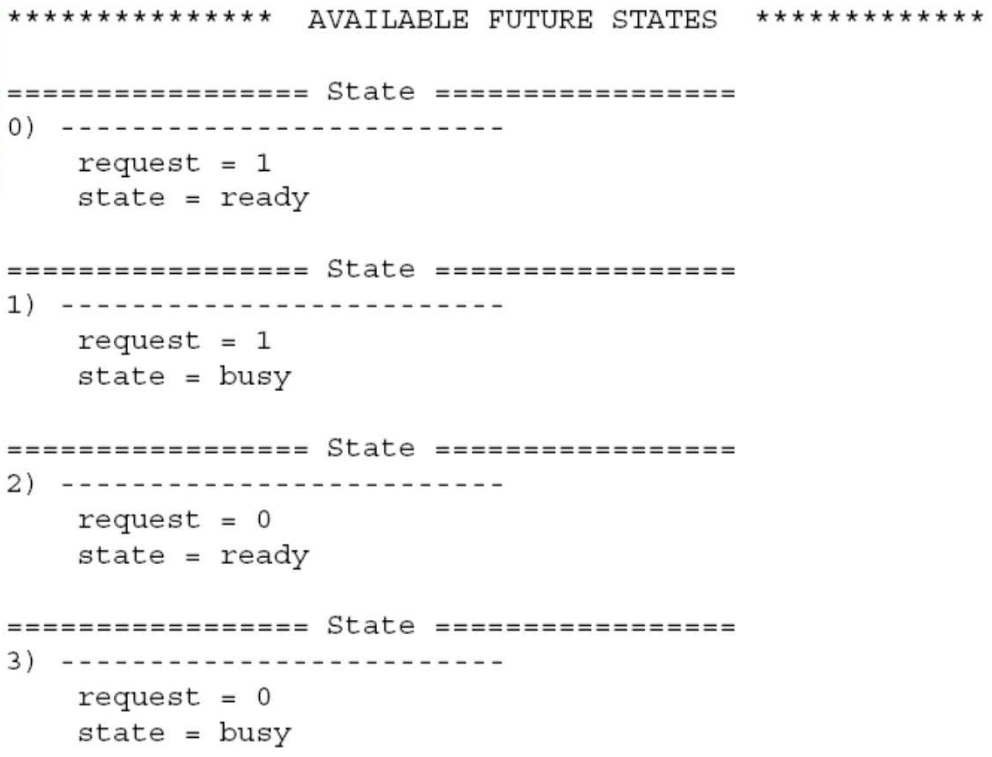
```
system_prompt> NuSMV -int short.smv
NuSMV> go
NuSMV> pick_state -r
NuSMV> print_current_state -v
Current state is 1.1
request = 0
state = ready
NuSMV> simulate -r 3
*********  Starting Simulation From State  1.1  *********
NuSMV> show_traces -t
There is 1 trace currently available.
NuSMV> show_traces -v
#################### Trace number: 1 ####################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
    request = 0
    state = ready
-> State: 1.2 <-
    request = 1
    state = busy
```

nuSMV: traces

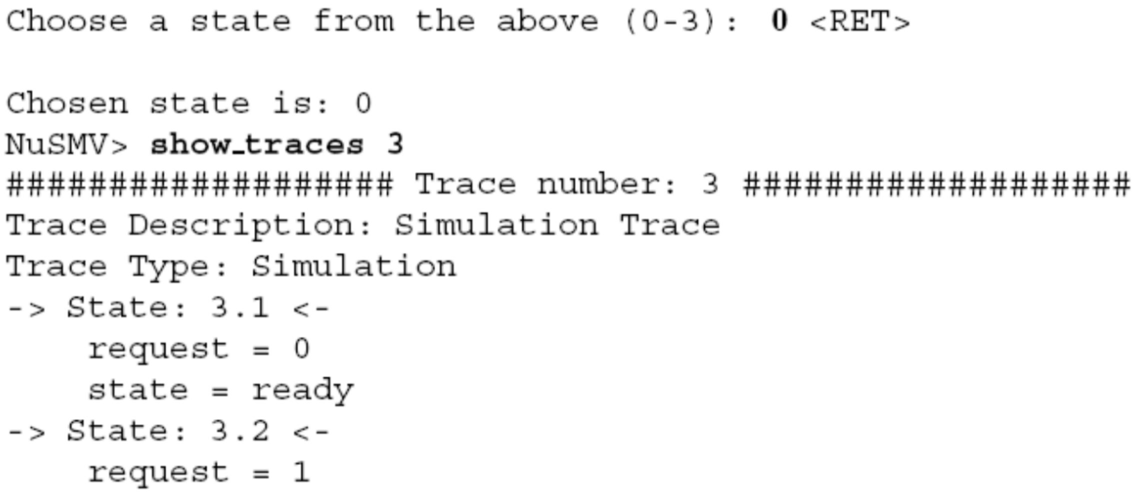
```
-> State: 1.3 <-
    request = 1
    state = ready
-> State: 1.4 <-
    request = 1
    state = busy
NuSMV> goto_state 1.4
The starting state for new trace is:
-> State 2.4 <-
    request = 1
    state = busy
NuSMV> simulate -r 3
********  Simulation Starting From State  2.4  ********
NuSMV> show_traces 2
################### Trace number: 2 ###################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 2.1 <-
    request = 1
    state = ready
-> State: 2.2 <-
    state = busy
-> State: 2.3 <-
    request = 0
-> State: 2.4 <-
    request = 1
-> State: 2.5 <-
    request = 0
-> State: 2.6 <-
    state = ready
-> State: 2.7 <-
NuSMV>
```

nuSMV: traces – choose a state

```
NuSMV> pick_state -i

***************  AVAILABLE STATES  *************

================= State =================
0) -------------------------
     request = 1
     state = ready

================= State =================
1) -------------------------
     request = 0
     state = ready


Choose a state from the above (0-1): 1 <RET>

Chosen state is: 1
NuSMV> simulate -i 1
********  Simulation Starting From State  3.1  ********
```



```
**************  AVAILABLE FUTURE STATES  *************

================= State =================
0) -------------------------
     request = 1
     state = ready

================= State =================
1) -------------------------
     request = 1
     state = busy

================= State =================
2) -------------------------
     request = 0
     state = ready

================= State =================
3) -------------------------
     request = 0
     state = busy
```



```
Choose a state from the above (0-3):  0 <RET>

Chosen state is: 0
NuSMV> show_traces 3
################### Trace number: 3 ###################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 3.1 <-
    request = 0
    state = ready
-> State: 3.2 <-
    request = 1
```

nuSMV: CTL/LTL model checking

A CTL specification is a CTL formula preceeded by the keyword SPEC

A LTL specification is a LTL formula preceeded by the keyword LTLSPEC

If the formula is not true a trace that provides a counter example is shown (for A quantifiers, since for existential does not make sense)

nuSMV: CTL model checking of semaphore

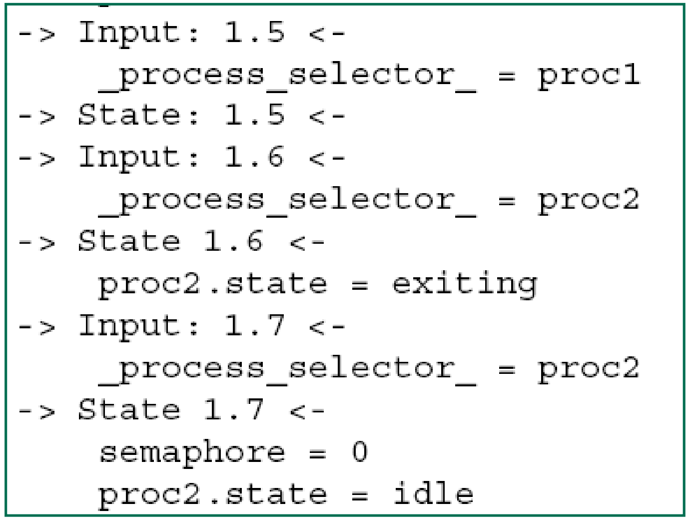
```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
 SPEC AG ! (proc1.state = critical & proc2.state = critical)
 SPEC AG (proc1.state = entering -> AF proc1.state = critical)
MODULE user(semaphore)
 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;
       1                             : state;
     esac;
   next(semaphore) :=
     case
       state = entering : 1;
       state = exiting  : 0;
       1                : semaphore;
     esac;
 FAIRNESS
   running
```

nuSMV: CTL model checking

```
system_prompt> NuSMV semaphore.smv
```

we obtain the following output:

```
-- specification AG (!(proc1.state = critical & proc2.state = critical))
-- is true
-- specification AG (proc1.state = entering -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following execution sequence
-> State: 1.1 <-
    semaphore = 0
    proc1.state = idle
    proc2.state = idle
-> Input: 1.2 <-
    _process_selector_ = proc1
-- Loop starts here
-> State: 1.2 <-
    proc1.state = entering
-> Input: 1.3 <-
    _process_selector_ = proc2
-> State: 1.3 <-
    proc2.state = entering
-> Input: 1.4 <-
    _process_selector_ = proc2
-> State: 1.4 <-
    semaphore = 1
    proc2.state = critical
```



```
-> Input: 1.5 <-
    _process_selector_ = proc1
-> State: 1.5 <-
-> Input: 1.6 <-
    _process_selector_ = proc2
-> State 1.6 <-
    proc2.state = exiting
-> Input: 1.7 <-
    _process_selector_ = proc2
-> State 1.7 <-
    semaphore = 0
    proc2.state = idle
```

CTL: Syntax

AP, set of atomic proposition. p∈AP.

CTL formulae:

$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid E[\varphi\, U\, \varphi] \mid A[\varphi\, U\, \varphi]$

E: “for some path”

A: “for all paths”

EX: “for some path next”

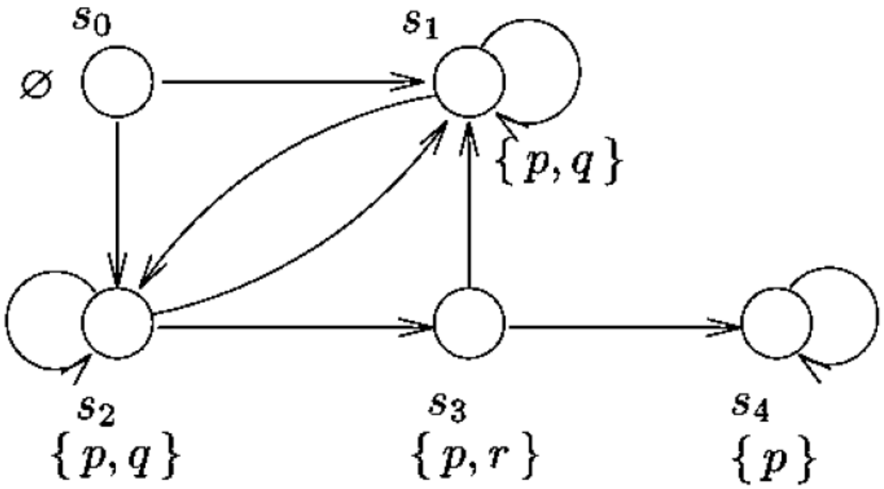
U: until

Note: syntactically correct formulas quantifiers and temporal operators are in strict alternation

Derived operators

- EFφ ≡ E[true U φ] “φ holds potentially” - “φ is possible”
- AFφ ≡ A[true U φ] “φ is inevitable (unavoidable)”
- EGφ ≡ ¬AF¬φ “potentially always φ” – "globally along some path"
- AGφ ≡ ¬EF¬φ “invariantly φ”
- AXφ ≡ ¬EX¬φ “for all paths next”

Implementare in NuSMV

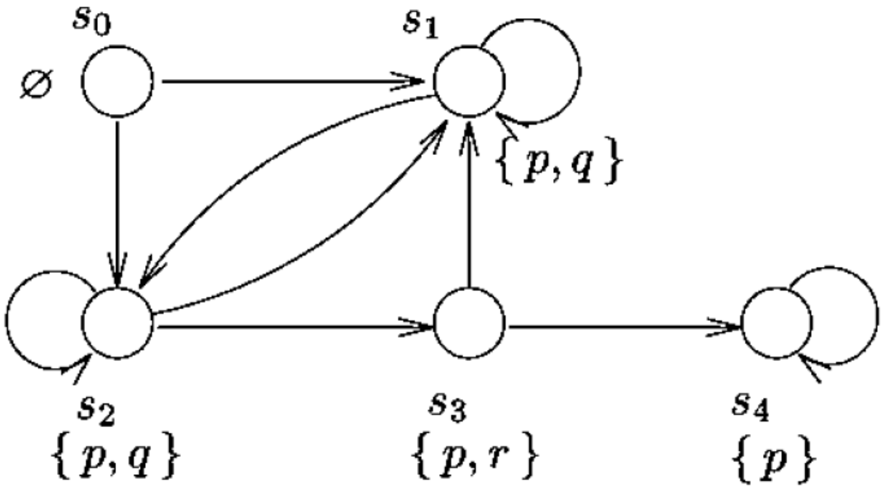


1. $\mathsf{EG}\, p$
2. $\mathsf{AG}\, p$
3. $\mathsf{EF}\, [\mathsf{AG}\, p]$
4. $\mathsf{AF}\, [p \,\mathsf{U}\, \mathsf{EG}\, (p \Rightarrow q)]$
5. $\mathsf{EG}\, [((p \wedge q) \vee r) \,\mathsf{U}\, (r \,\mathsf{U}\, \mathsf{AG}\, p)]$

Check the validity of the formulae in each state

EF$\varphi \equiv$ E[true U φ] "φ holds potentially"
AF$\varphi \equiv$ A[true U φ] "φ is inevitable"
EG$\varphi \equiv \neg$AF$\neg\varphi$ "potentially always φ"
AG$\varphi \equiv \neg$EF$\neg\varphi$ "invariantly φ"

Exercise on CTL – implement in nuSMV



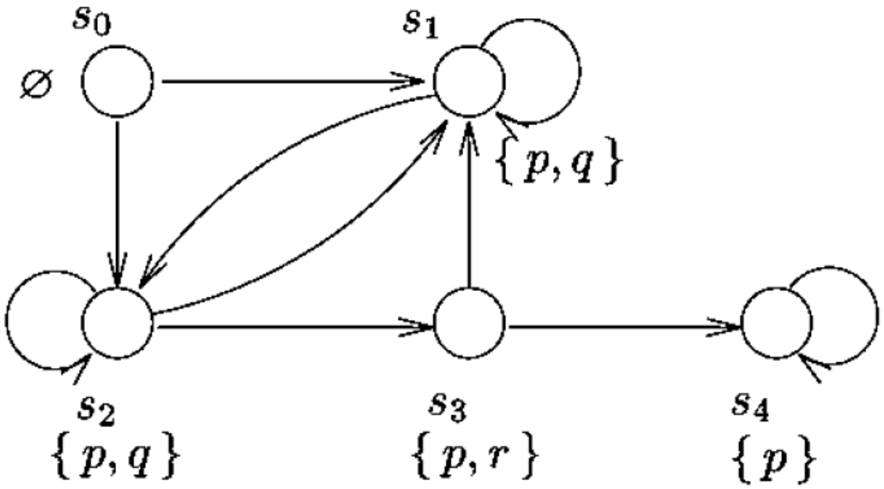
$EFp \equiv E[\text{true } U\ p]$

$AFp \equiv A[\text{true } U\ p]$

EFp: start with Q = {s1, s2, s3, s4} and in one step add s0, and at the next iteration the algorithm stops

AFp: start with Q = {s1, s2, s3, s4} and in the next step consider s0. S0 can be added only if all arcs out of s0 are in Q

Exercise on CTL – implement in nuSMV



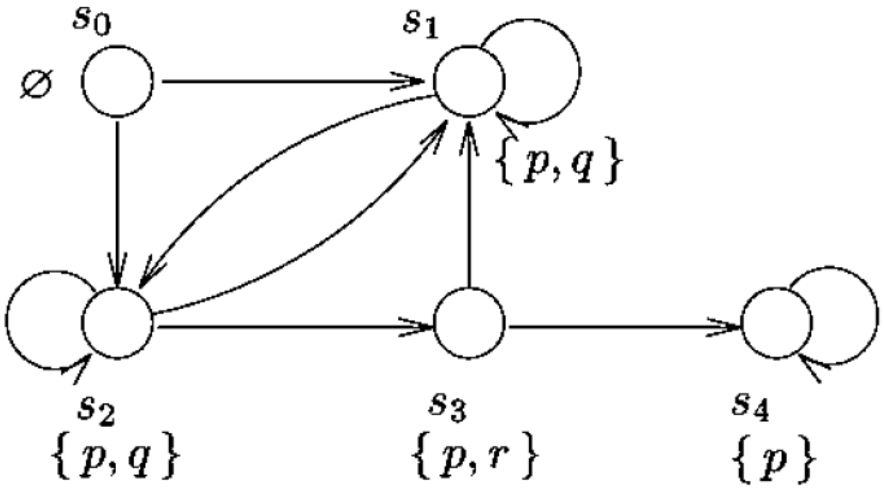
EGp ≡¬AF¬p ≡ ¬A[true U ¬p]

AGp ≡ ¬EF¬p ≡ ¬E[true U ¬p]

EGp: the result is the complement of the states that satisfy AF¬p that can be computed as before

AGp: the result is the complement of the states that satisfy EF¬p

Exercise on CTL – implement in nuSMV



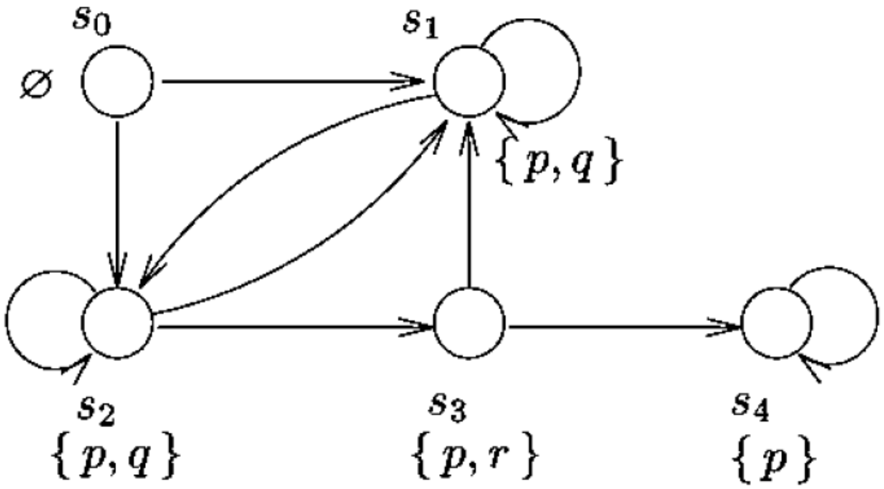
EFq ≡ E[true U q]

AFq ≡ A[true U q]

EFq: start with Q = {s1, s2} and in one step add s0, and s3, and at the next iteration the algorithm stops

AFq: start with Q = {s1, s2} and in the next step s0 is added. At the next iteration no new element is added and the algorithm stops.

Exercise on CTL – implement in nuSMV



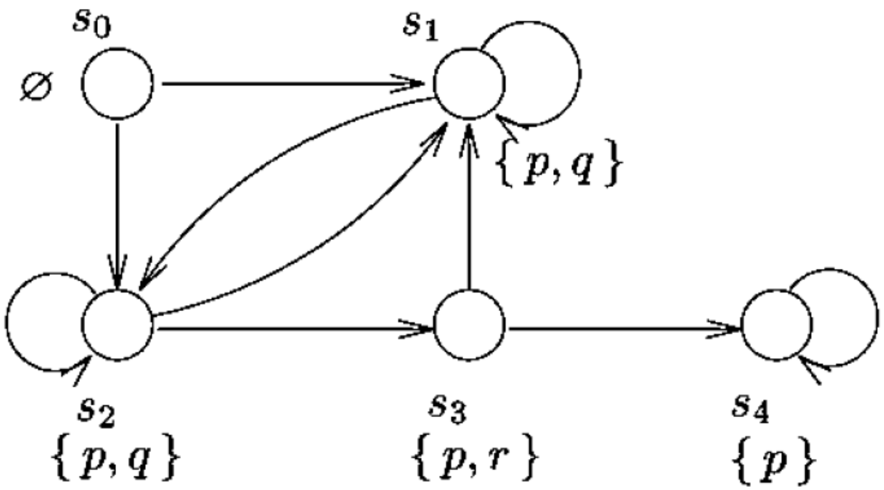
EGq ≡¬AF¬q ≡ E[true U q]

AGq ≡ ¬EF¬q ≡A[true U q]

EGq: the result is the complement of the states that satisfy AF¬q that can be computed as before

AGq: the result is the complement of the states that satisfy EF¬q

Exercise on CTL – implement in nuSMV



1. $\mathsf{EG}\, p$
2. $\mathsf{AG}\, p$
3. $\mathsf{EF}\, [\mathsf{AG}\, p]$
4. $\mathsf{AF}\, [p\, \mathsf{U}\, \mathsf{EG}\, (p \Rightarrow q)]$
5. $\mathsf{EG}\, [((p \wedge q) \vee r)\, \mathsf{U}\, (r\, \mathsf{U}\, \mathsf{AG}\, p)]$

E
A

Check the validity of the formulae in each state

End of nuSMV