



Model Checking

A Hands-On Introduction

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

13th 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 features 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)

The course at a glance



- Introduction
 - Introduction to Formal Methods and Model Checking
 - Models for Reactive Systems: Kripke Structures
 - Properties of Reactive Systems: CTL, LTL
- NuSMV model checker
 - The NuSMV Open Source project
 - The SMV language
 - NuSMV demo
- Advanced Topics
 - Symbolic Model Checking Techniques: BDD-based and SAT-based techniques
 - The SMV language (advanced)
- Play with NuSMV



Part 1 - Course Overview

– *Model Checking* –

A Hands-On Introduction

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

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

The Need for Formal Methods



- Development of Industrial Systems:
 - System reliability depends on the “correct” functioning of hardware and software.
 - Safety-critical, money-critical systems.
 - Growing complexity of environment and required functionalities.
 - Market issues: time-to-market, development costs.

The Need for Formal Methods (II)



- Difficulties with traditional methodologies:
 - Ambiguities (in requirements, architecture, detailed design).
 - Errors is specification/design refinements.
 - The assurance provided by testing can be too limited.
 - Testing came too late in development.
- Consequences:
 - Expensive errors in the early design phases.
 - Infeasibility of achieving high reliability requirements.
 - Low software quality: limited modifiability.

Formal Methods: Solution and Benefits

- The problem:
 - Certain (sub)systems can be too complicated/critical to design with traditional techniques.
- Formal Methods:
 - *Formal Specification*: precise, unambiguous description.
 - *Formal Validation & Verification Tools*: exhaustive analysis of the formal specification.
- Potential Benefits:
 - Find design bugs in early design stages.
 - Achieve higher quality standards.
 - Shorten time-to-market reducing manual validation phases.
 - Produce well documented, maintainable products.

Highly recommended by ESA, NASA for the design of safety-critical systems.

Formal Methods: Potential Problems



- Main Issue: Effective use of Formal Methods
 - debug/verify during the design process;
 - without slowing down the design process;
 - without increasing costs too much.
- Potential Problems of Formal Methods:
 - formal methods can be too costly;
 - formal methods can be not effective;
 - training problems;
 - the verification problem can be too difficult.
- How can we get benefits and avoid these problems?
 - by adapting our technologies and tools to the specific problem at hand;
 - using “automated” verification techniques (e.g. model checking).

Formal Verification Techniques



- Model-based simulation or testing
 - method: test for ϕ by exploring possible behaviors.
 - tools: test case generators.
 - applicable if: the system defines an executable model.
- Deductive Methods
 - method: provide a formal proof that ϕ holds.
 - tool: theorem prover, proof assistant or proof checker.
 - applicable if: systems can be described as a mathematical theory.
- Model Checking
 - method: systematic check of ϕ in all states of the system.
 - tools: model checkers.
 - applicable if: system generates (finite) behavioral model.

Simulation and Testing



- Basic procedure:
 - take a model (simulation) or a realization (testing).
 - stimulate it with certain inputs, i.e. the test cases.
 - observe produce behavior and check whether this is “desired”.
- Drawback:
 - The number of possible behaviors can be too large (or even infinite).
 - Unexplored behaviors may contain the fatal bug.
- Testing and simulation can show the presence of bugs, *not their absence*.

Theorem Proving



- Basic procedure:
 - describe the system as a mathematical theory.
 - express the property in the mathematical theory.
 - prove that the property is a theorem in the mathematical theory.
- Drawback:
 - Express the system as a mathematical theory can be difficult.
 - Find a proof can require a big effort.
- Theorem proving can be used to prove *absence of bugs*.

Model Checking



- Basic procedure:
 - describe the system as Finite State Model.
 - express properties in Temporal Logic.
 - formal V&V by automatic exhaustive search over the state space.
- Drawback:
 - State space explosion.
 - Expressivity – hard to deal with parametrized systems.
- Model checking can be used to prove *absence of bugs*.

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 development stages (cfr. Pentium bug);
 - exhaustive, thus effective (often bugs are also in scaled-down problems).
 - provides counterexamples (directs the designer to the problem).

Model Checking in a nutshell



- Reactive systems represented as a finite state models (in this course, Kripke models).
- System behaviors represented as (possibly) infinite sequences of states.
- Requirements represented as formulae in temporal logics.
- “*The system satisfies the requirement*” represented as truth of the formula in the Kripke model.
- Efficient model checking algorithms based on exhaustive exploration of the Kripke model.

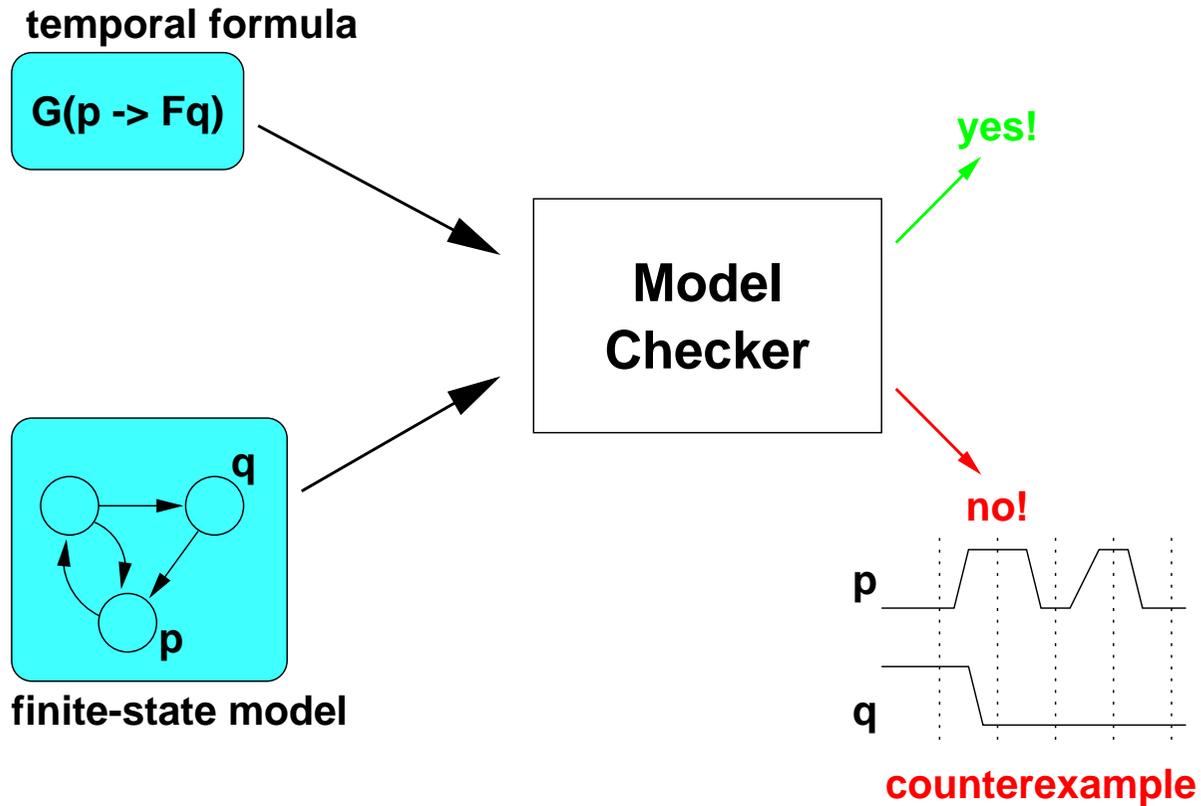
What is a Model Checker



A model checker is a software tool that

- given a description of a Kripke model M ...
- ... and a property Φ ,
- decides whether $M \models \Phi$,
- returns “yes” if the property is satisfied,
- otherwise returns “no”, and provides a counterexample.

What is a Model Checker



What is not covered in this course



- A deep theoretical background. We will focus on practice.
- Advanced model checking techniques:
 - abstraction;
 - compositional, assume-guarantee reasoning;
 - symmetry reduction;
 - approximation techniques (e.g. directed to bug hunting);
 - model transformation techniques (e.g. minimization wrt to bisimulation).
- Many other approaches and tools for model checking.
- Model checking for infinite-state (e.g. hybrid, timed) systems.



Part 2 - Symbolic Model Checking

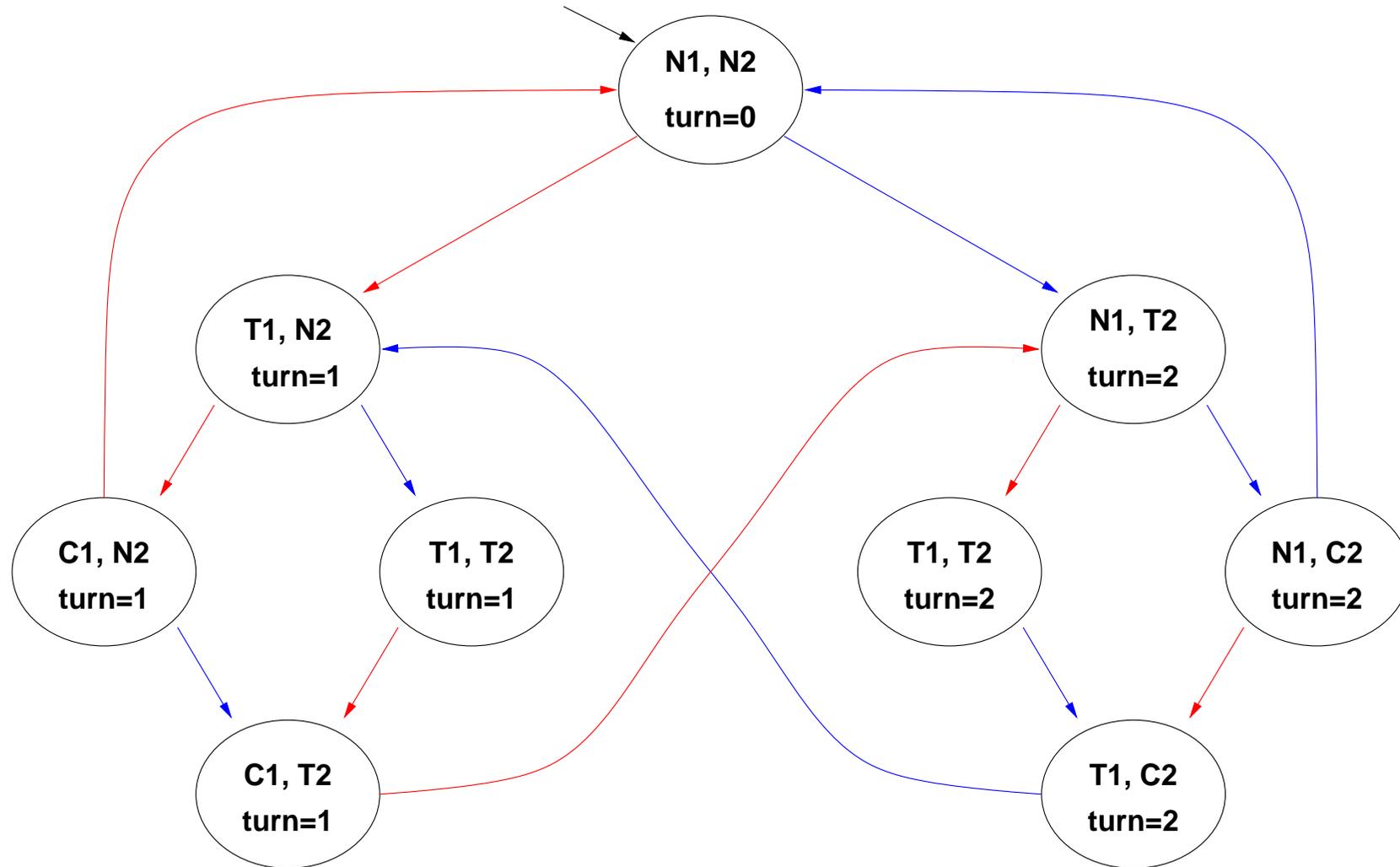
– *Model Checking* –

A Hands-On Introduction

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

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

A Kripke model for mutual exclusion

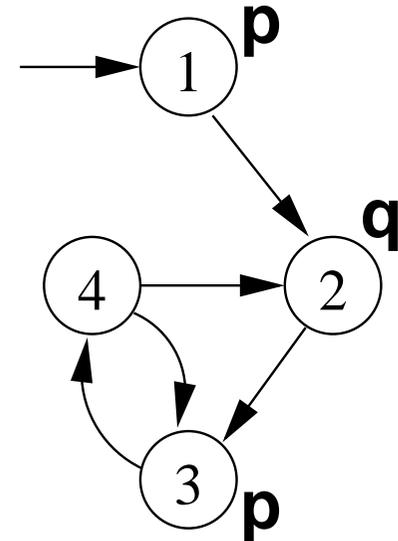


N = noncritical, T = trying, C = critical User 1 User 2

Modeling the system: Kripke models



- Kripke models are used to describe reactive systems:
 - nonterminating systems with infinite behaviors,
 - e.g. communication protocols, operating systems, hardware circuits;
 - represent dynamic evolution of modeled systems;
 - values to state variables, program counters, content of communication channels.
- Formally, a Kripke model (S, R, I, AP, L) consists of
 - a set of states S ;
 - a set of initial states $I \subseteq S$;
 - a set of transitions $R \subseteq S \times S$;
 - a set of atomic propositions AP ;
 - a labeling $L \subseteq S \times AP$.



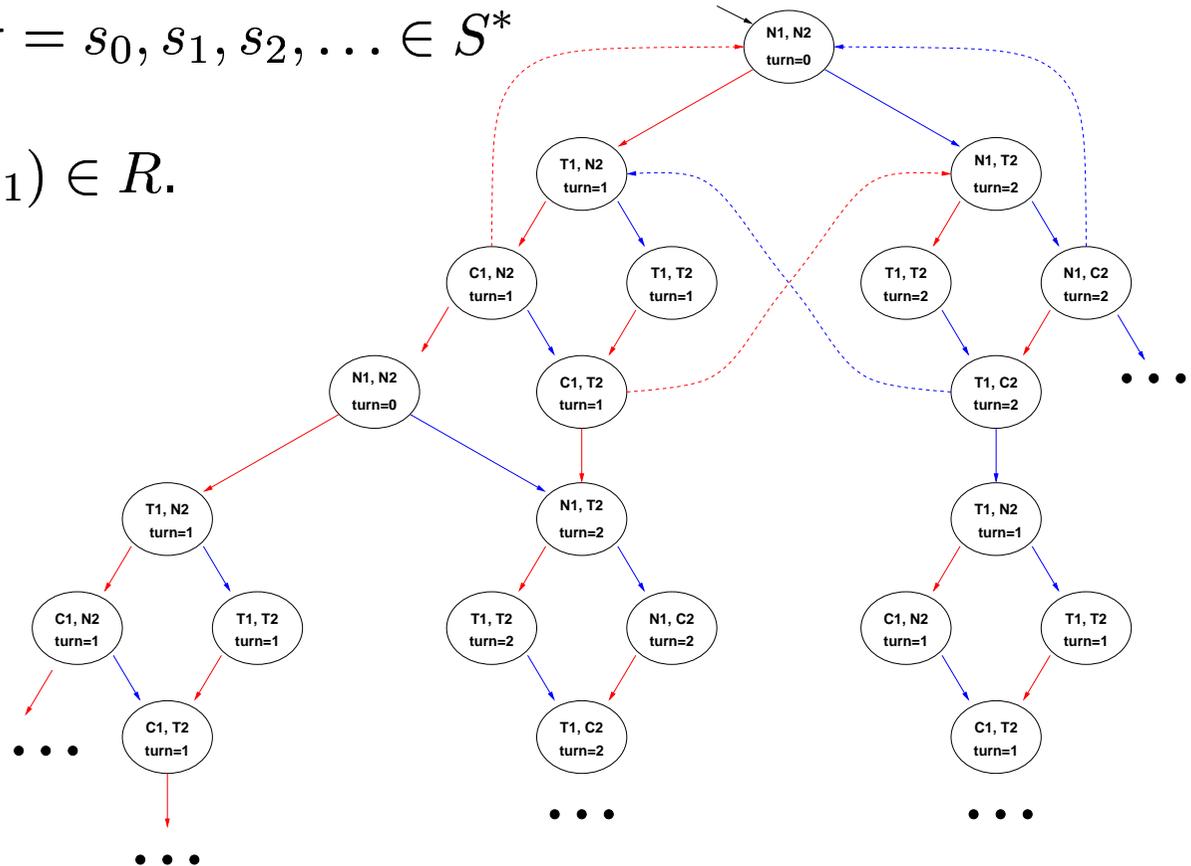
Path in a Kripke Model



- A path in a Kripke model M is an infinite sequence

$$\sigma = s_0, s_1, s_2, \dots \in S^*$$

such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$.



- A state s is reachable in M if there is a path from the initial states to s .

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.

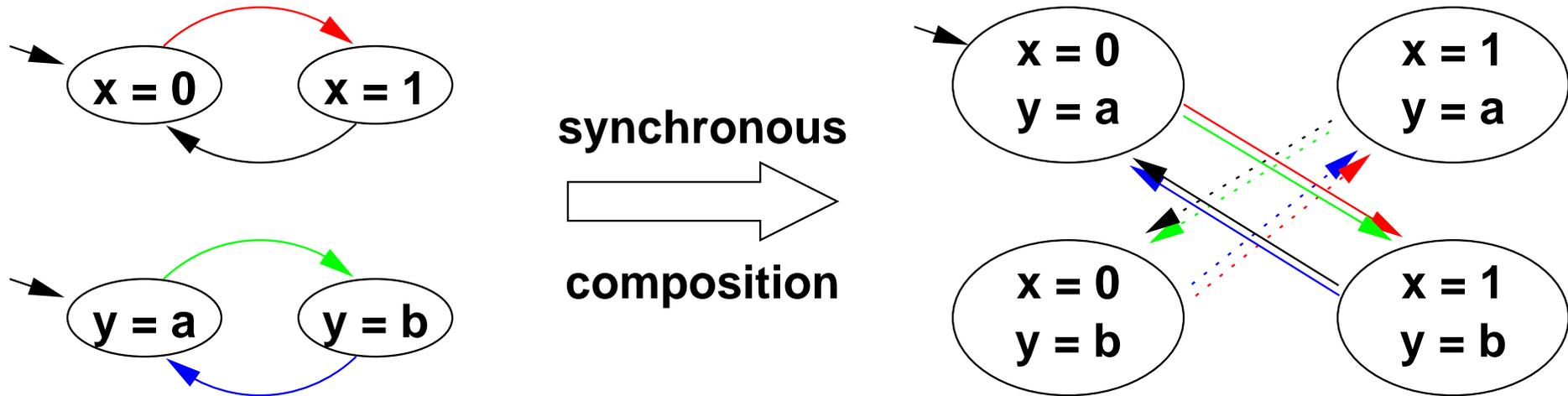
State explosion problem in model checking:

- 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.

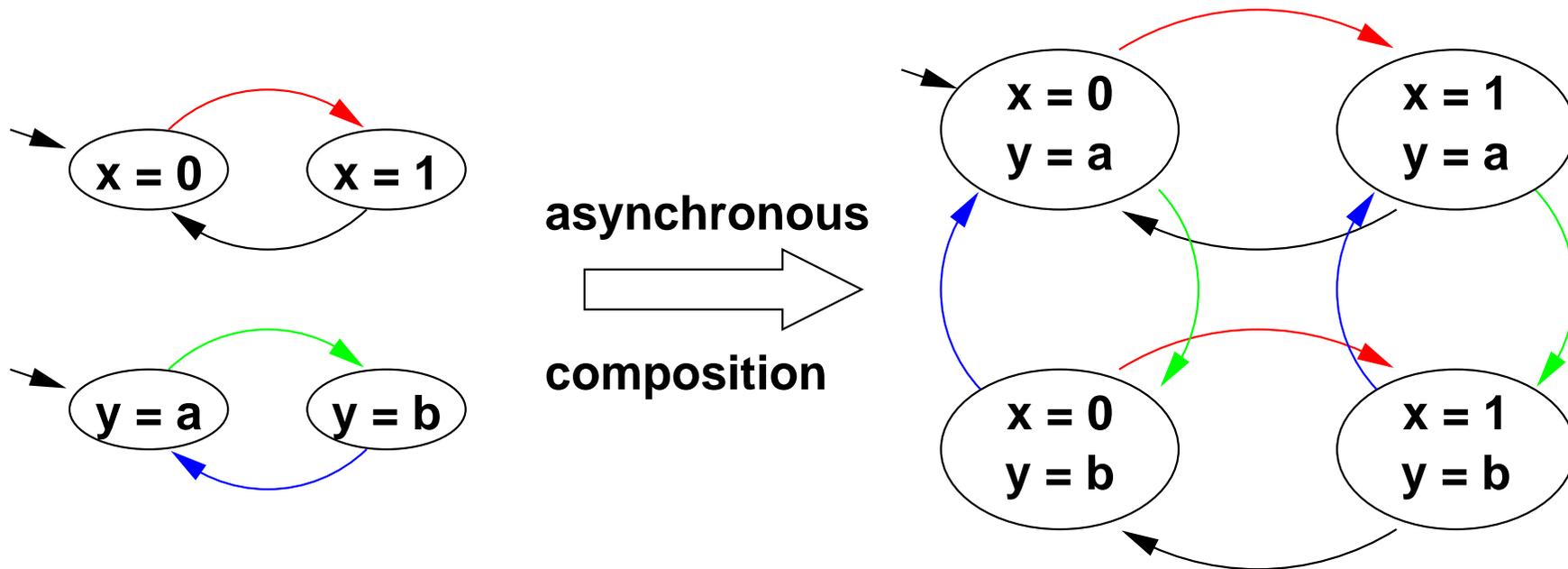


- Typical example: sequential hardware circuits.
- Synchronous composition is the default in NuSMV.

Asynchronous Composition



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



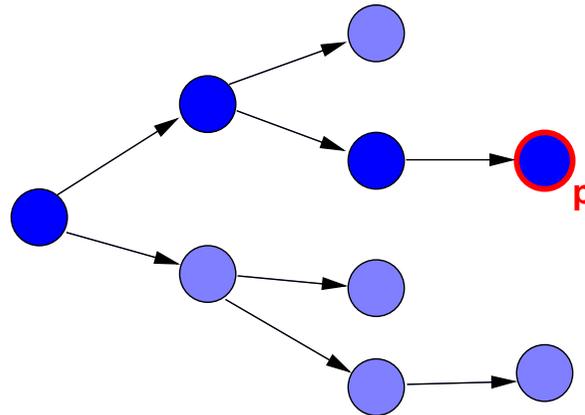
- Typical example: communication protocols.
- Asynchronous composition can be represented with NuSMV processes.

Properties of Reactive Systems (I)



Safety properties:

- nothing bad ever happens
 - deadlock: two processes waiting for input from each other, the system is unable to perform a transition.
 - no reachable state satisfies a “bad” condition, e.g. never two processes in critical section at the same time
- can be refuted by a finite behaviour
- it is never the case that p .

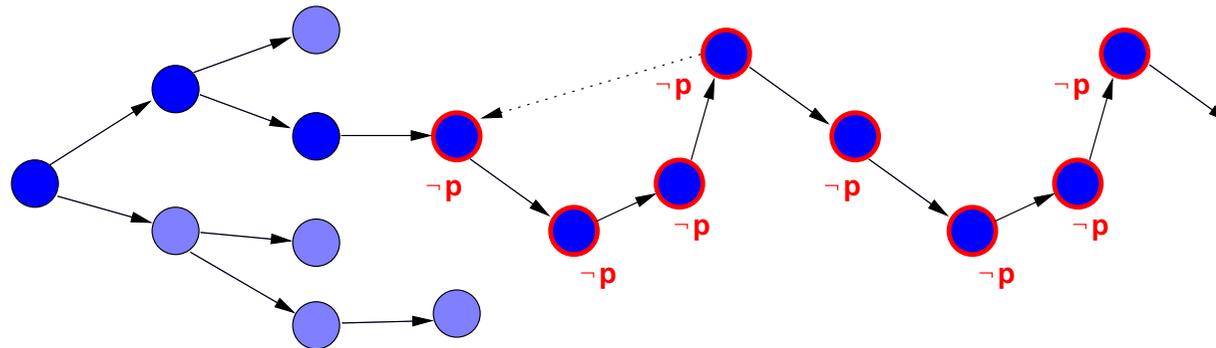


Properties of Reactive Systems (II)



Liveness properties:

- Something desirable will eventually happen
 - whenever a subroutine takes control, it will always return it (sooner or later)
- can be refuted by infinite behaviour
 - a subroutine takes control and never returns it



- an infinite behaviour can be presented as a loop

Temporal Logics



- Express properties of “Reactive Systems”
 - nonterminating behaviours,
 - without explicit reference to time.
- Linear Time Temporal Logic (LTL)
 - interpreted over each path of the Kripke structure
 - linear model of time
 - temporal operators
- Computation Tree Logic (CTL)
 - interpreted over computation tree of Kripke model
 - branching model of time
 - temporal operators plus path quantifiers

Linear Time Temporal Logic (LTL)



LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \dots \rightarrow s[t] \rightarrow s[t + 1] \rightarrow \dots$$

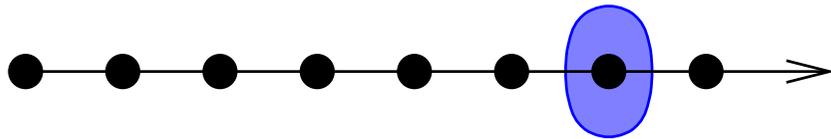
LTL provides the following temporal operators:

- “Finally” (or “future”): Fp is true in $s[t]$ iff p is true in **some** $s[t']$ with $t' \geq t$
- “Globally” (or “always”): Gp is true in $s[t]$ iff p is true in **all** $s[t']$ with $t' \geq t$
- “Next”: Xp is true in $s[t]$ iff p is true in $s[t + 1]$
- “Until”: pUq is true in $s[t]$ iff
 - q is true in some state $s[t']$ with $t' \geq t$
 - p is true in all states $s[t'']$ with $t \leq t'' < t'$

LTL

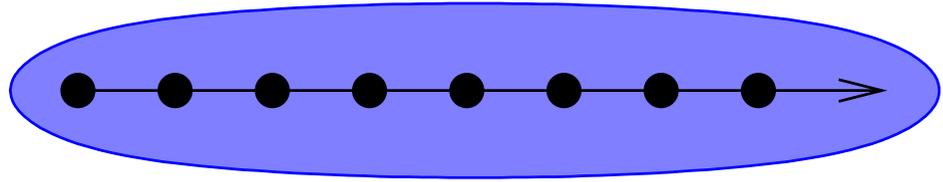


finally P



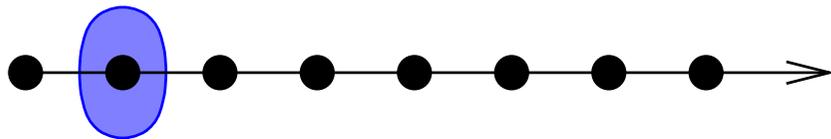
$F P$

globally P



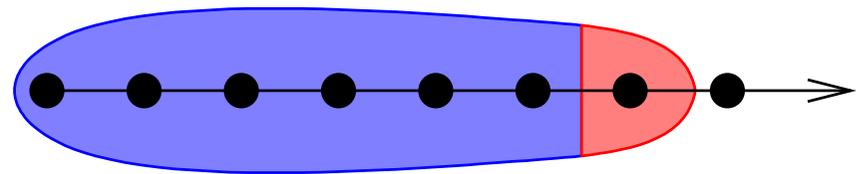
$G P$

next P



$X P$

P until q



$P U q$

LTL: Examples



- Liveness: “if input, then eventually output”

$$G(\text{input} \rightarrow F\text{output})$$

- Strong fairness: “infinitely send implies infinitely recv.”

$$GF\text{send} \rightarrow GF\text{recv}$$

- Weak until: “no output before input”

$$\neg\text{output } W \text{ input}$$

$$\text{where } p W q \leftrightarrow (p U q \vee Gp)$$

Computation Tree Logic (CTL)

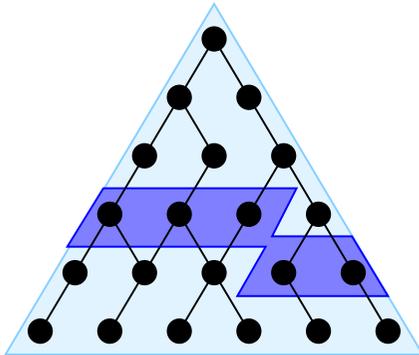


- CTL properties are evaluated over trees.
- Every temporal operator (F, G, X, U) preceded by a path quantifier (A or E).
- Universal modalities (AF, AG, AX, AU): the temporal formula is true in **all** the paths starting in the current state.
- Existential modalities (EF, EG, EX, EU): the temporal formula is true in **some** of the paths starting in the current state.

CTL

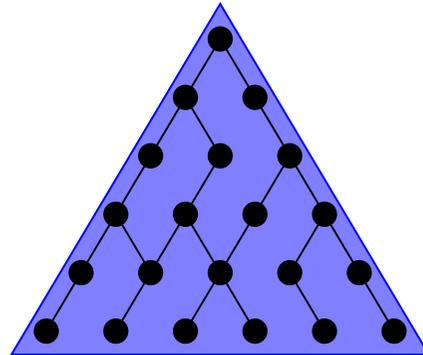


finally P



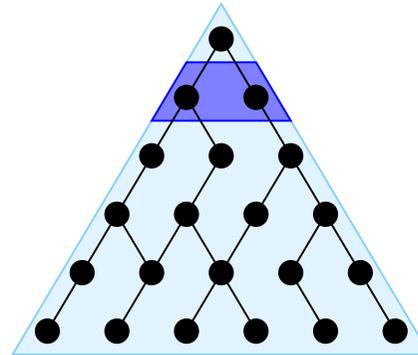
$AF P$

globally P



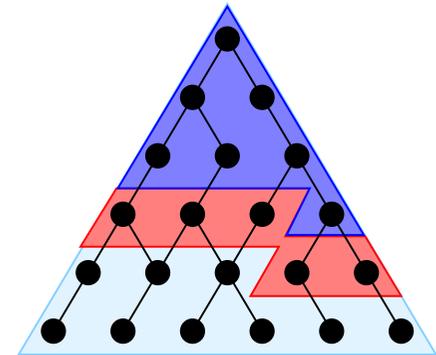
$AG P$

next P

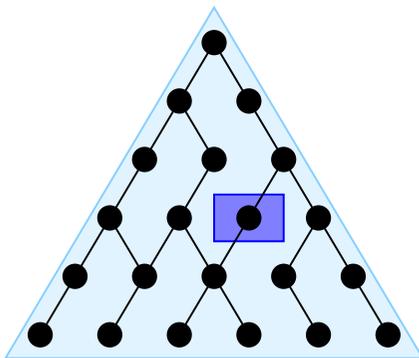


$AX P$

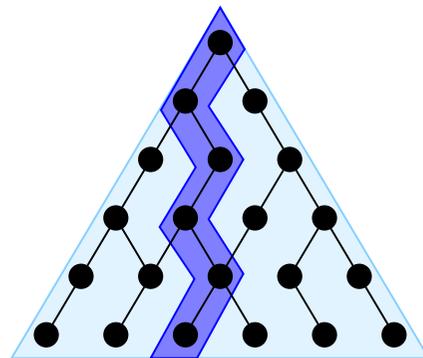
P until q



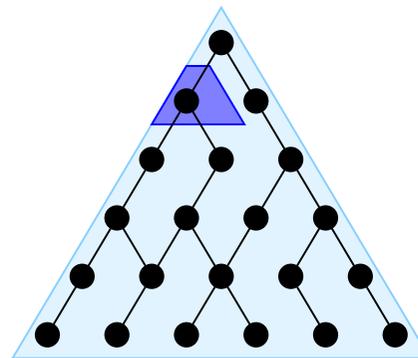
$A[P U q]$



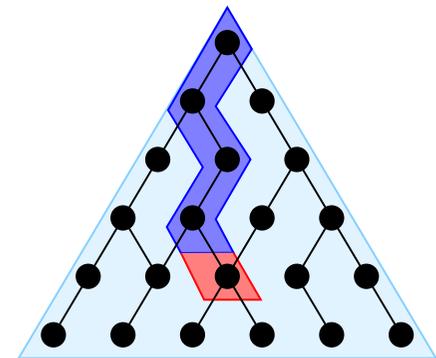
$EF P$



$EG P$



$EX P$



$E[P U q]$

CTL



- Some dualities:

$$AGp \leftrightarrow \neg EF\neg p$$

$$AFp \leftrightarrow \neg EG\neg p$$

$$AXp \leftrightarrow \neg EX\neg p$$

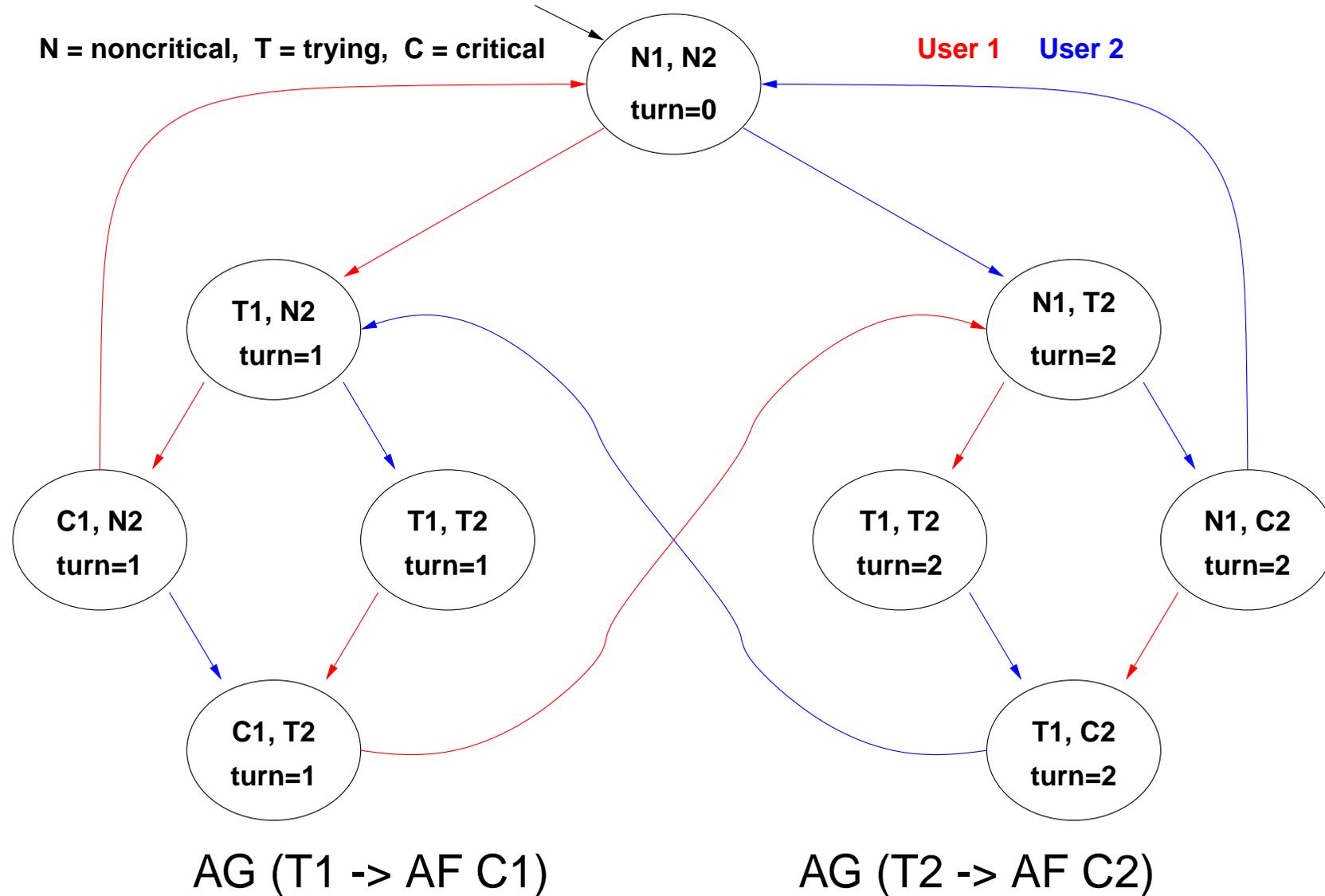
- Example: specifications for the mutual exclusion problem.

$AG\neg(C_1 \wedge C_2)$ mutual exclusion

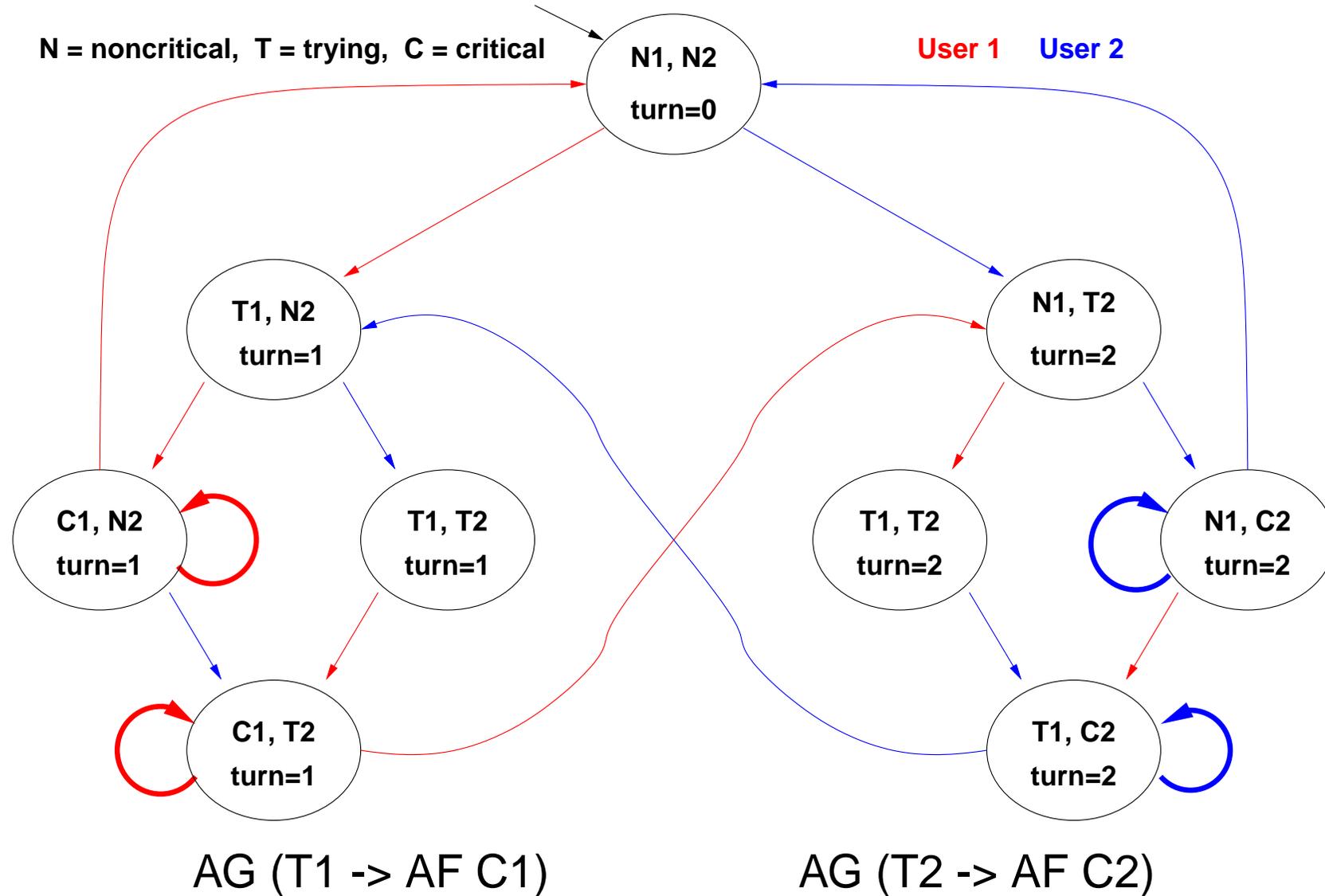
$AG(T_1 \rightarrow AF C_1)$ liveness

$AG(N_1 \rightarrow EX T_1)$ non-blocking

The need for fairness conditions



The need for fairness conditions



Fair Kripke models



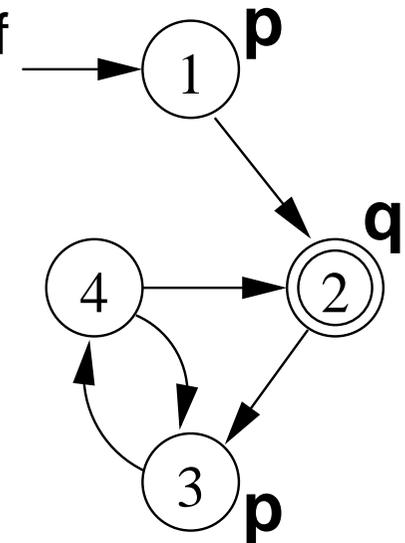
- Intuitively, fairness conditions are used to eliminate behaviours where a condition never holds
 - e.g. once a process is in critical section, it never exits

● Formally, a Fair Kripke model (S, R, I, AP, L, F) consists of

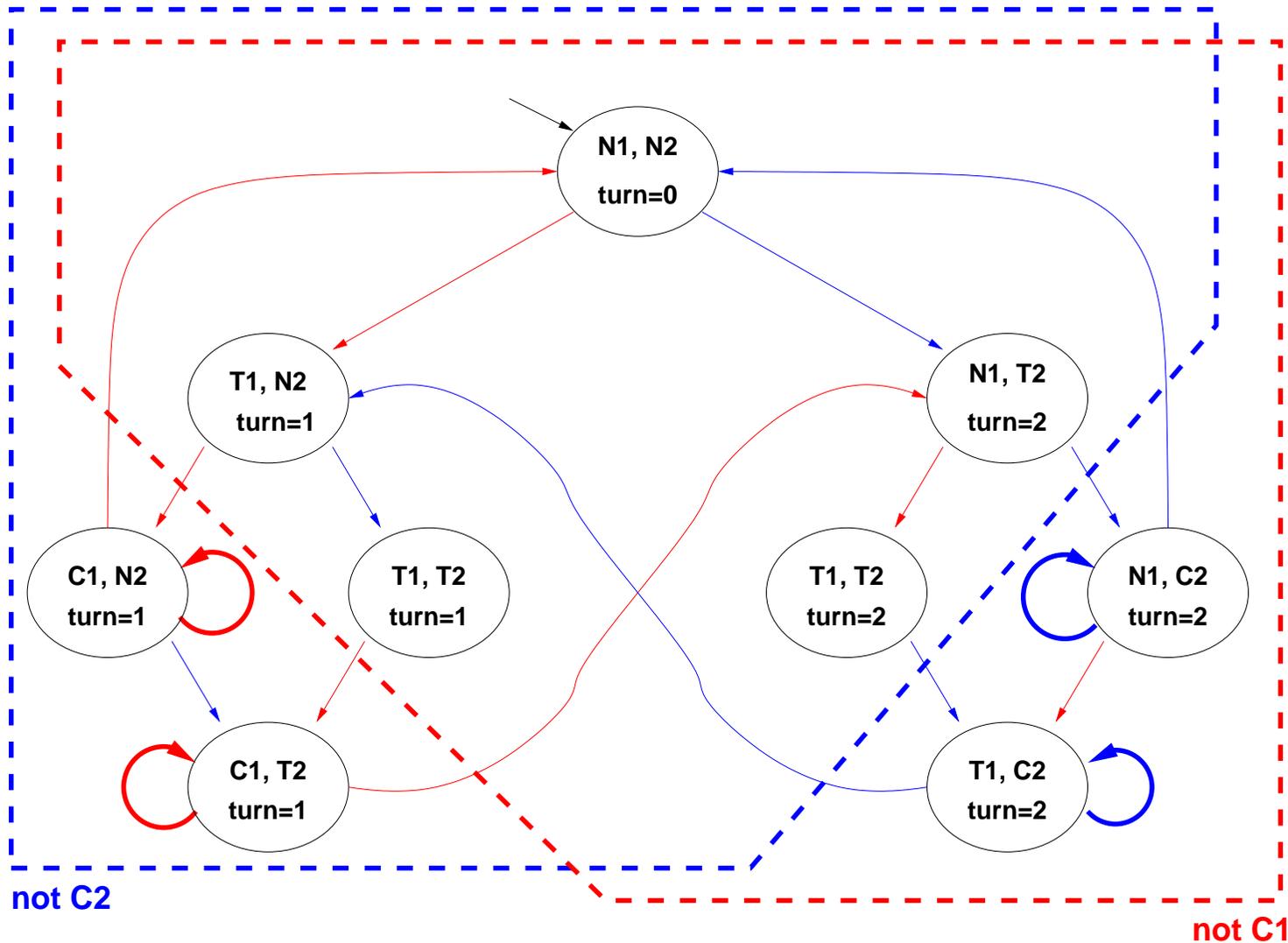
- a set of states S ;
- a set of initial states $I \subseteq S$;
- a set of transitions $R \subseteq S \times S$;
- a set of atomic propositions AP ;
- a labeling $L \subseteq S \times AP$;

\Rightarrow a set of fairness conditions $F = \{f_1, \dots, f_n\}$, with $f_i \subseteq S$.

- Fair path: at least one state for each f_i occurs in the path an infinite number of times.
- Fair state: a state from which at least one fair path originates.



Fairness: $\{\{\text{not } C1\}, \{\text{not } C2\}\}$

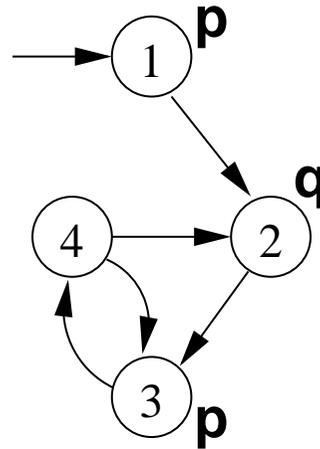


Model Checking



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: **$G(p \rightarrow Fq)$**

CTL: **$AG(p \rightarrow AFq)$**

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

The Main Problem: State Space Explosion



- The bottleneck:
 - Exhaustive analysis may require to store all the states of the Kripke structure
 - The state space may be exponential in the number of components
 - State Space Explosion: too much memory required
- Symbolic Model Checking:
 - Symbolic representation
 - Different search algorithms

Symbolic Model Checking

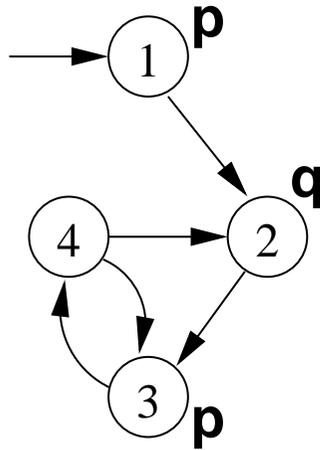


- Symbolic representation:
 - manipulation of *sets of states* (rather than single states);
 - sets of states represented by formulae in propositional logic;
 - set cardinality not directly correlated to size
 - expansion of *sets of transitions* (rather than single transitions);
 - two main symbolic techniques:
 - Binary Decision Diagrams (BDDs)
 - Propositional Satisfiability Checkers (SAT solvers)
- Different model checking algorithms:
 - Fix-point Model Checking (historically, for CTL)
 - Bounded Model Checking (historically, for LTL)
 - Invariant Checking

CTL Model Checking: Example



Consider a simple system and a specification:

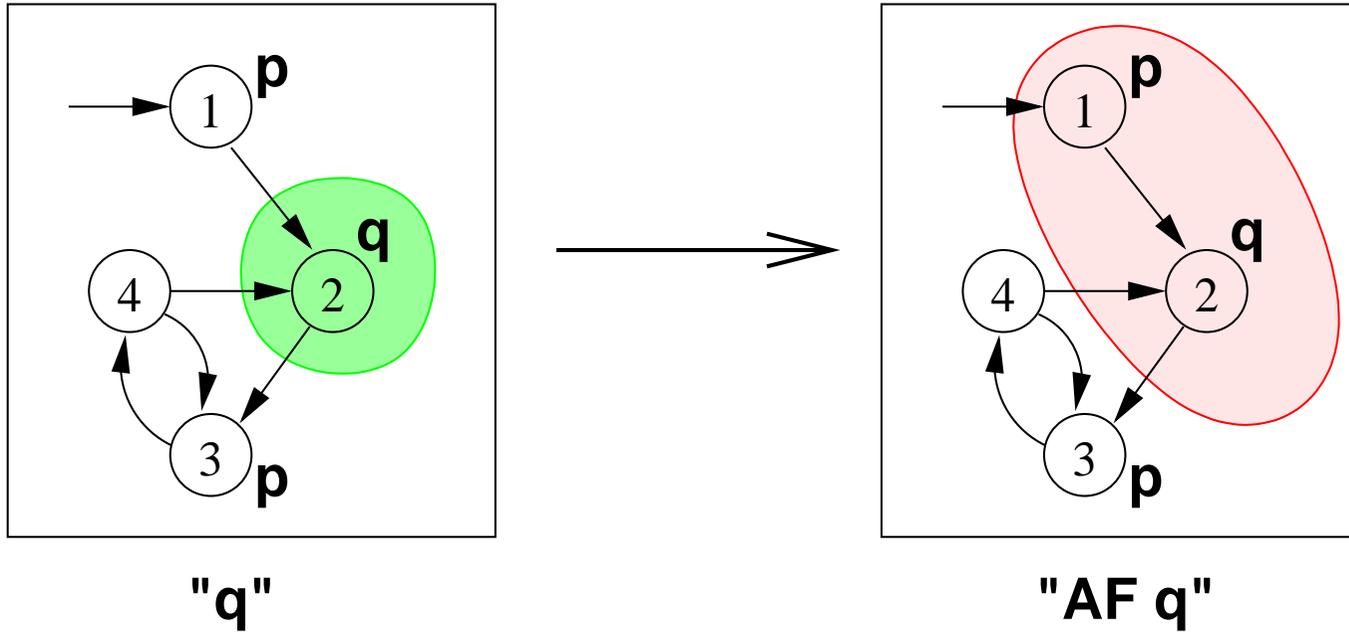


$AG(p \rightarrow AFq)$

Idea:

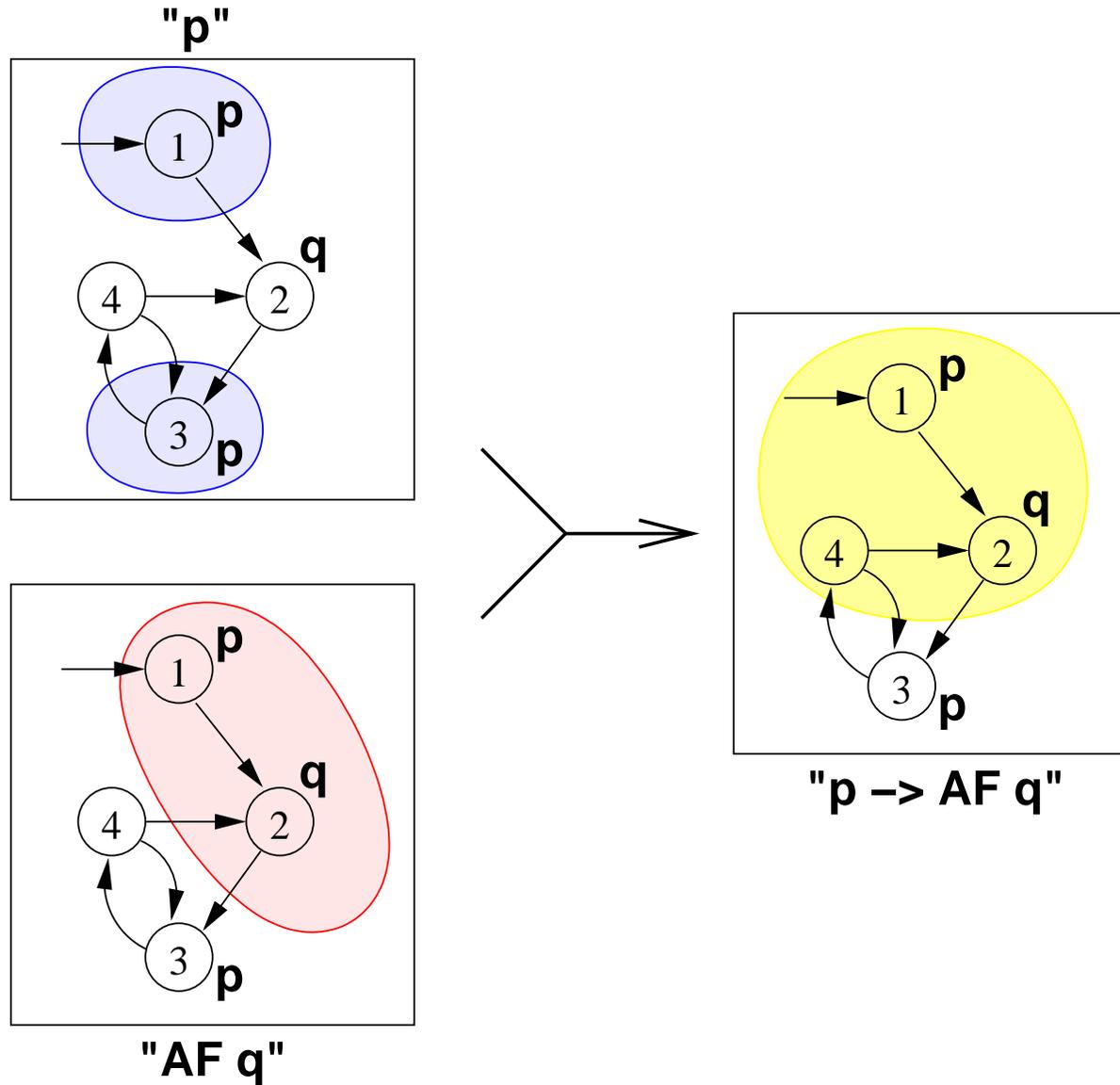
- construct the set of states where the formula holds
- proceeding “bottom-up” on the structure of the formula
- **$q, AFq, p, p \rightarrow AF q, AG(p \rightarrow AF q)$**

CTL Model Checking: Example

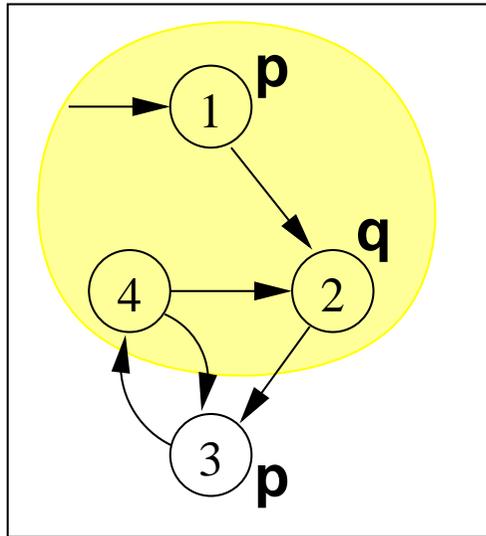


AF q is the union of **q**, **AX q**, **AX AX q**, ...

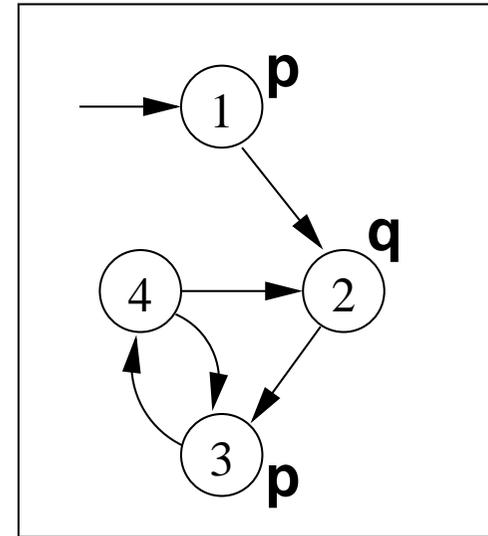
CTL Model Checking: Example



CTL Model Checking: Example



" $p \rightarrow AF q$ "



" $AG(p \rightarrow AF q)$ "

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 $AX \Phi$, apply preimage computation
 - $\forall s'. (\mathcal{T}(s, s') \rightarrow \Phi(s'))$
- on $AF \Phi$, compute least fixpoint using
 - $AF \Phi \leftrightarrow (\Phi \vee AX AF \Phi)$
- on $AG \Phi$, compute greatest fixpoint using
 - $AG \Phi \leftrightarrow (\Phi \wedge AX AG \Phi)$

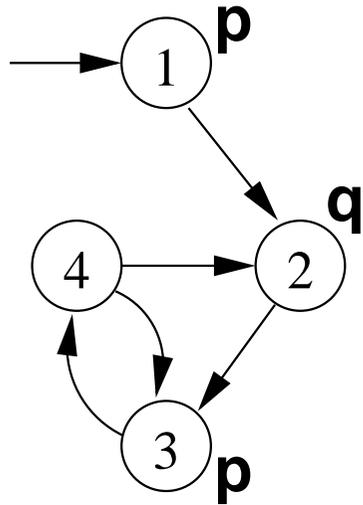
Bounded Model Checking

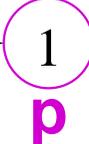


Key ideas:

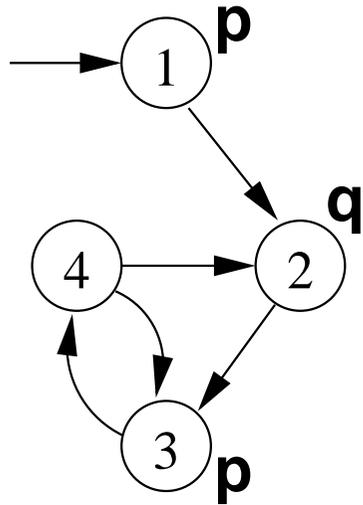
- looks for counter-example paths of increasing length k
 - oriented to finding bugs
- for each k , builds a boolean formula that is satisfiable iff there is a counter-example of length k
 - can be expressed using $k \cdot |s|$ variables
 - formula construction is not subject to state explosion
- satisfiability of the boolean formulas is checked using a *SAT procedure*
 - can manage complex formulae on several 100K variables
 - returns satisfying assignment (i.e., a counter-example)

Bounded Model Checking: Example

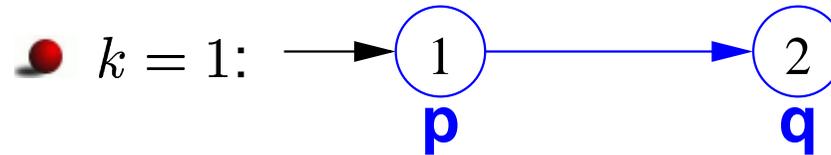


- Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- Negated Formula (violation): $\mathbf{F}(p \ \& \ \mathbf{G} \ ! \ q)$
- $k = 0$: 
- No counter-example found.

Bounded Model Checking: Example

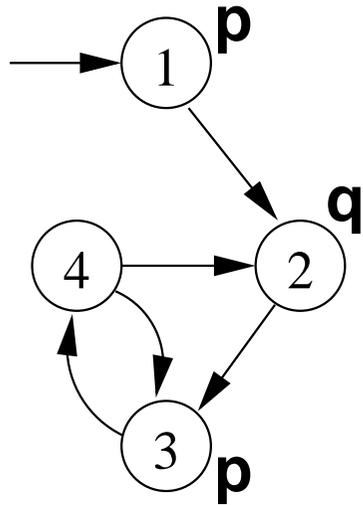


• Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$



• No counter-example found.

Bounded Model Checking: Example

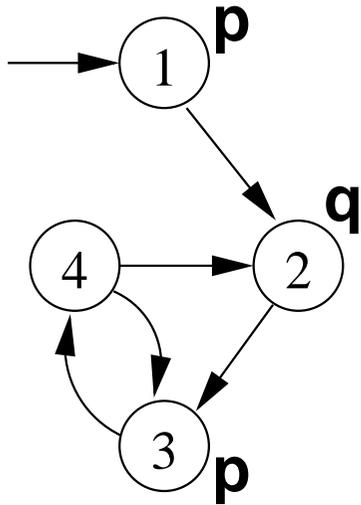


• Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$



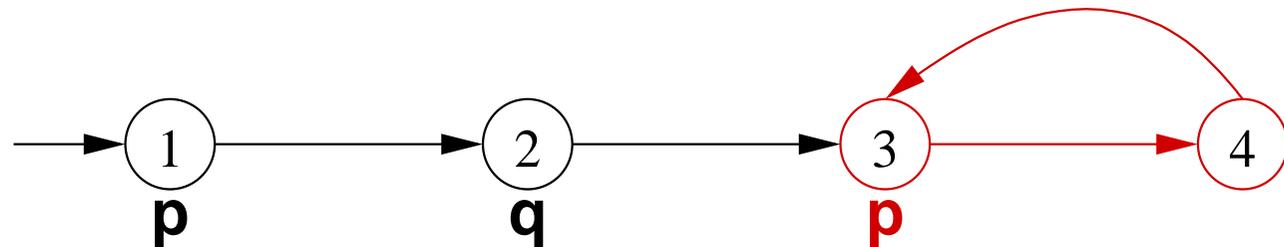
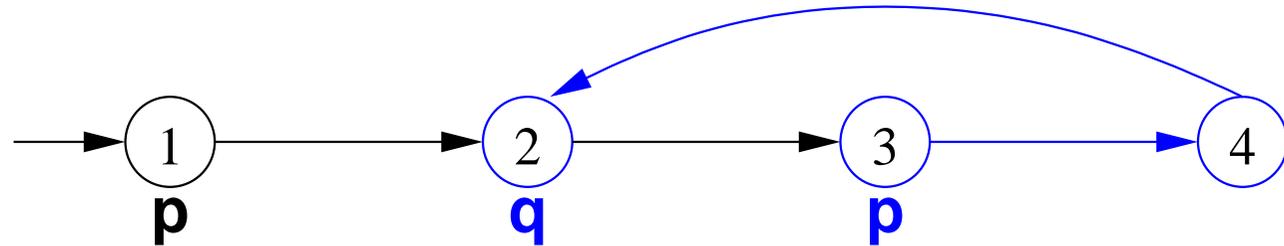
• No counter-example found.

Bounded Model Checking: Example



● Formula: $G(p \rightarrow Fq)$

● $k = 3$:



● The 2nd trace is a counter-example!

Bounded Model Checking



- *Bounded Model Checking:*

Given a FSM $\mathcal{M} = \langle S, \mathcal{I}, \mathcal{T} \rangle$, an LTL property ϕ and a bound $k \geq 0$:

$$\mathcal{M} \models_k \phi$$

- This is equivalent to the satisfiability problem on formula:

$$[[\mathcal{M}, \phi]]_k \equiv [[\mathcal{M}]]_k \wedge [[\phi]]_k$$

where:

- $[[\mathcal{M}]]_k$ is a k -path compatible with \mathcal{I} and \mathcal{T} :

$$\mathcal{I}(s_0) \wedge \mathcal{T}(s_0, s_1) \wedge \dots \wedge \mathcal{T}(s_{k-1}, s_k)$$

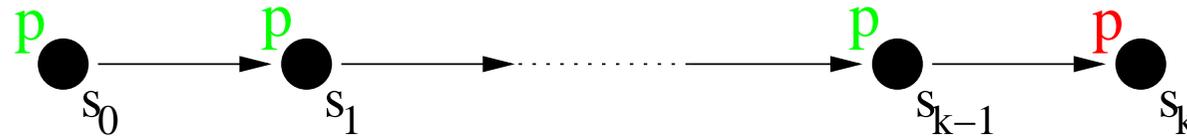
- $[[\phi]]_k$ says that the k -path satisfies ϕ

Bounded Model Checking: Examples



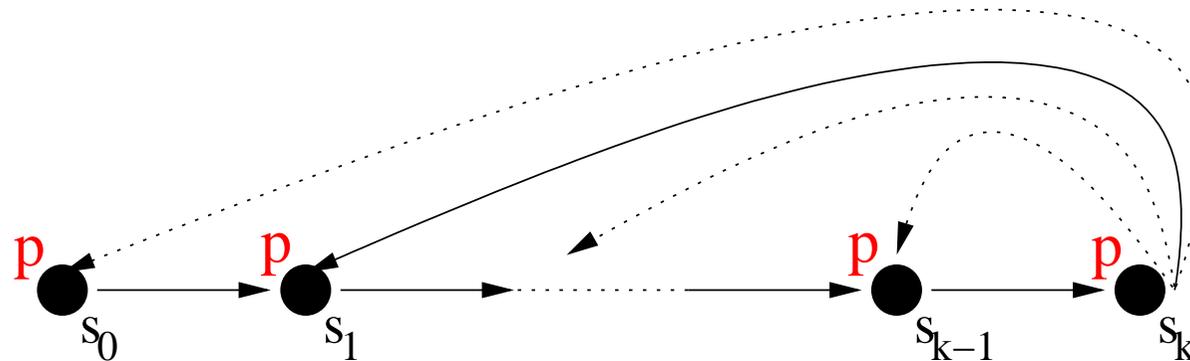
● $\phi = F p$

$$\llbracket F p \rrbracket_k = \bigvee_{i=0}^k p(\mathbf{s}_i)$$



● $\phi = G p$

$$\llbracket G p \rrbracket_k = \bigvee_{i=0}^k \left(\mathcal{T}(\mathbf{s}_k, \mathbf{s}_i) \wedge \bigwedge_{i=0}^k p(\mathbf{s}_i) \right)$$

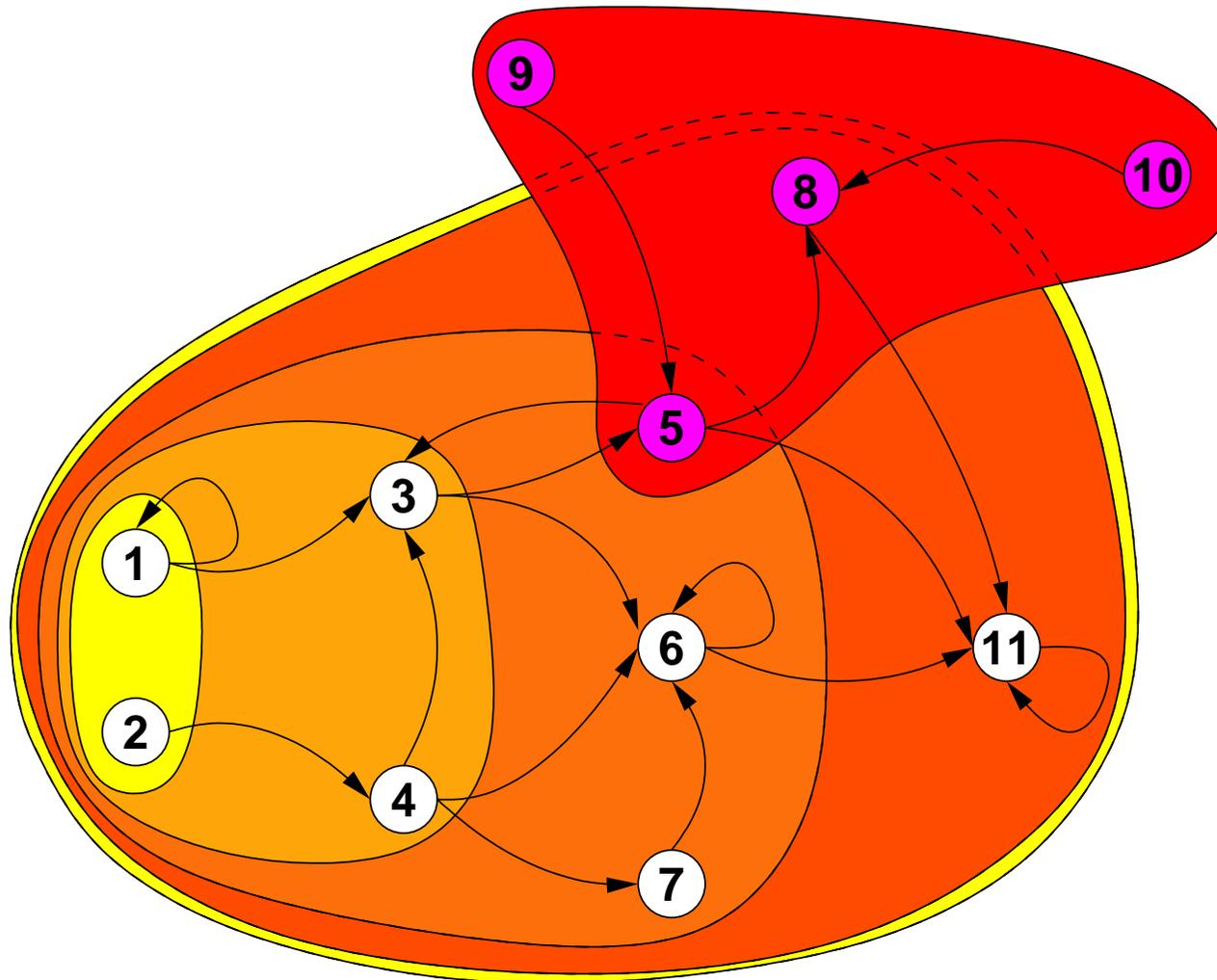


Symbolic Model Checking of Invariants



Checking invariant properties (e.g. **AG ! bad** is a reachability problem):

● is there a reachable state that is also a bad state (●)?

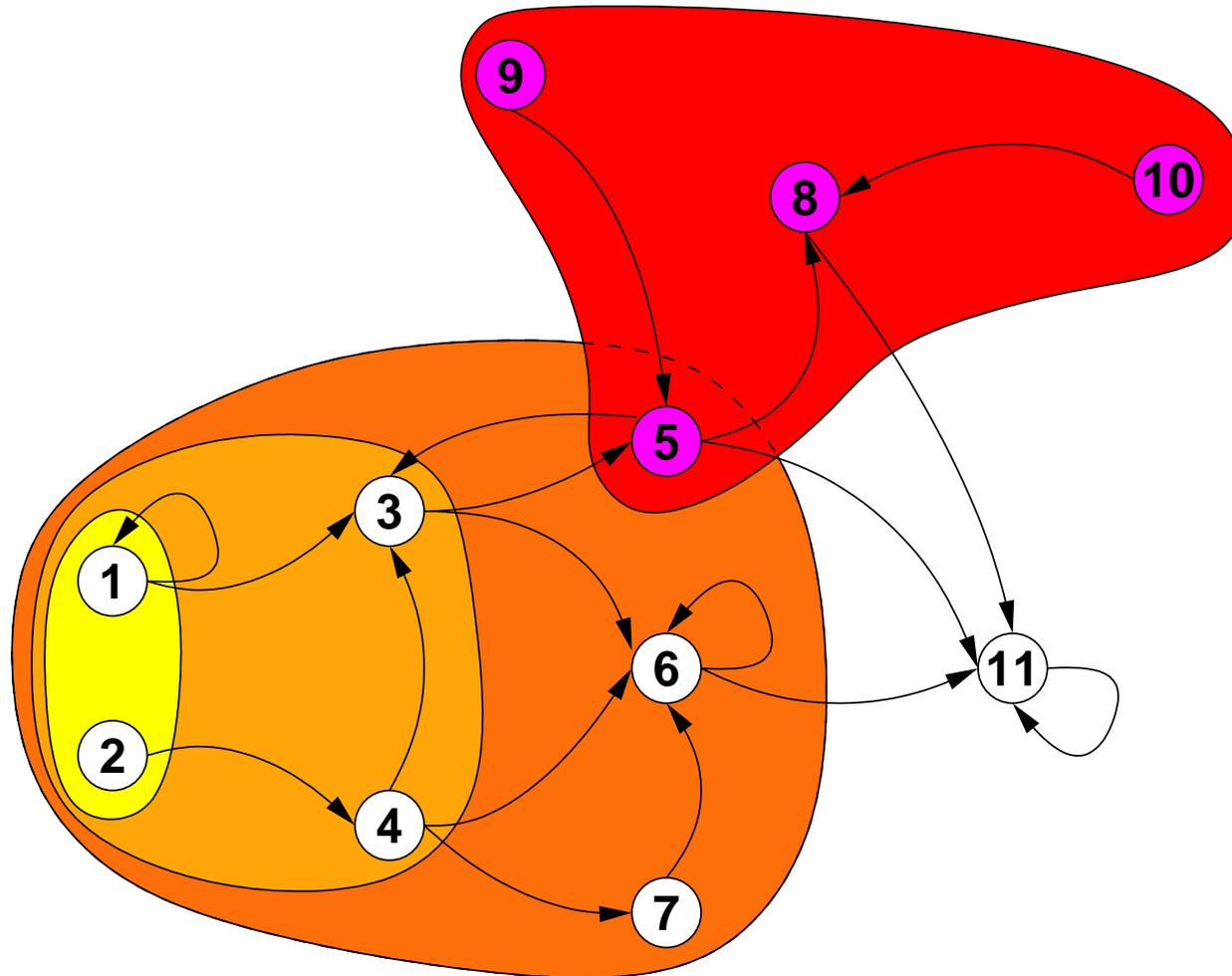


On the fly Checking of Invariants



Anticipate bug detection:

- at each layer, check if a new state is a bug

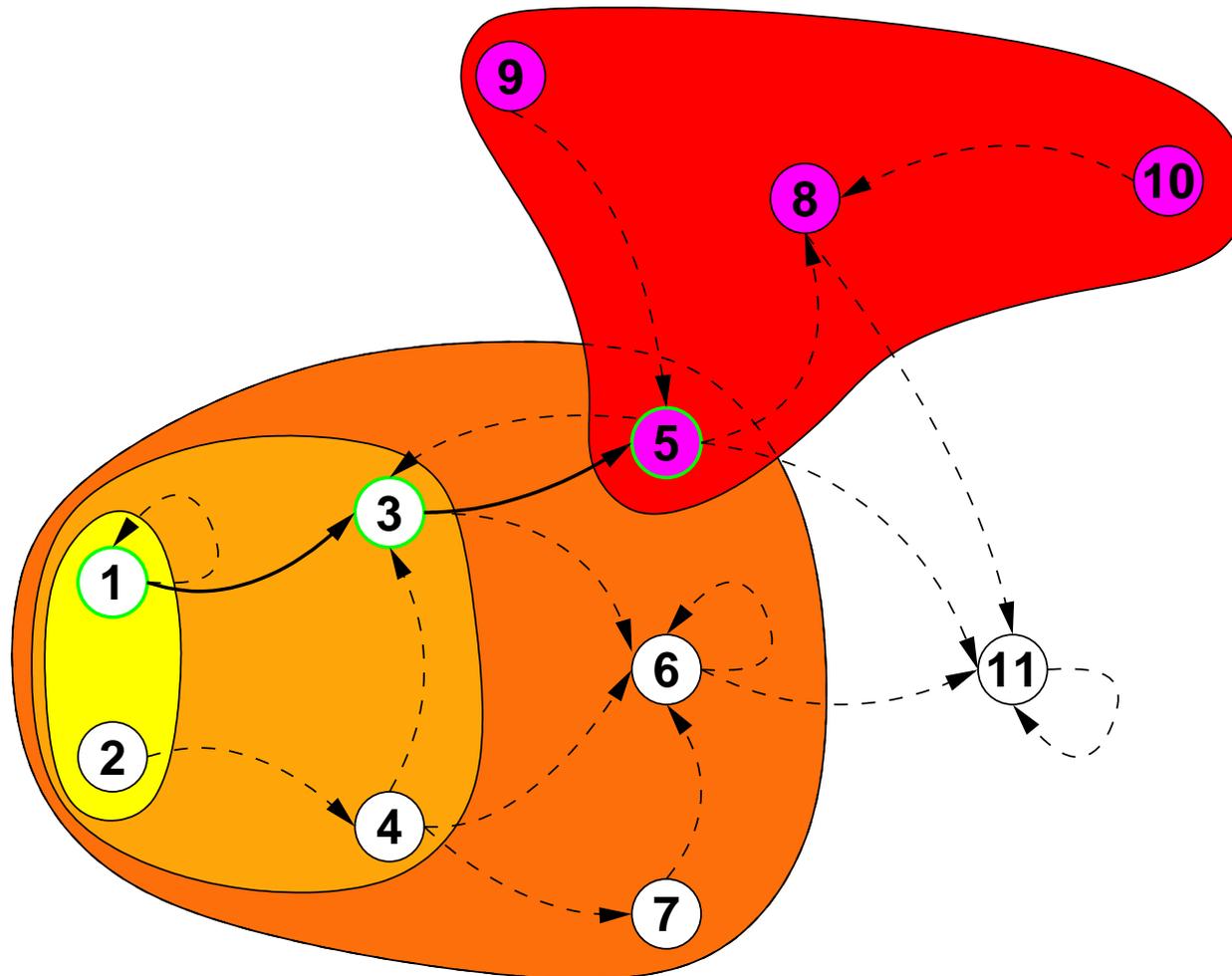


On the fly Checking of Invariants: Counterexamples



If a bug is found,

- a counterexample can be reconstructed proceeding backwards



Inductive Reasoning on Invariants



1. If all the initial states are good,
 2. and if from any good state we only go to good states
- ⇒ then we can conclude that the system is correct for all reachable states.



Part 3 - The NuSMV Model Checker

– *Model Checking*–

A Hands-On Introduction

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

13th 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)*:

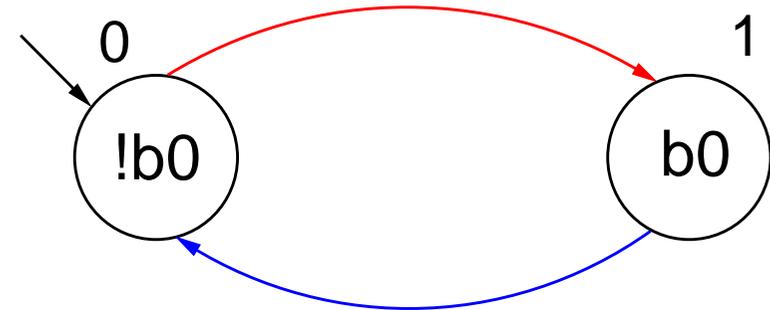
- anyone can freely download, copy, use, modify, and redistribute NuSMV 2
- any modification and extension should be made publicly available under the terms of LGPL (“copyleft”)

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 ($\text{init}(b0) := 0$).
- ➡ Assignments that define the transition relation ($\text{next}(b0) := !b0$).

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):

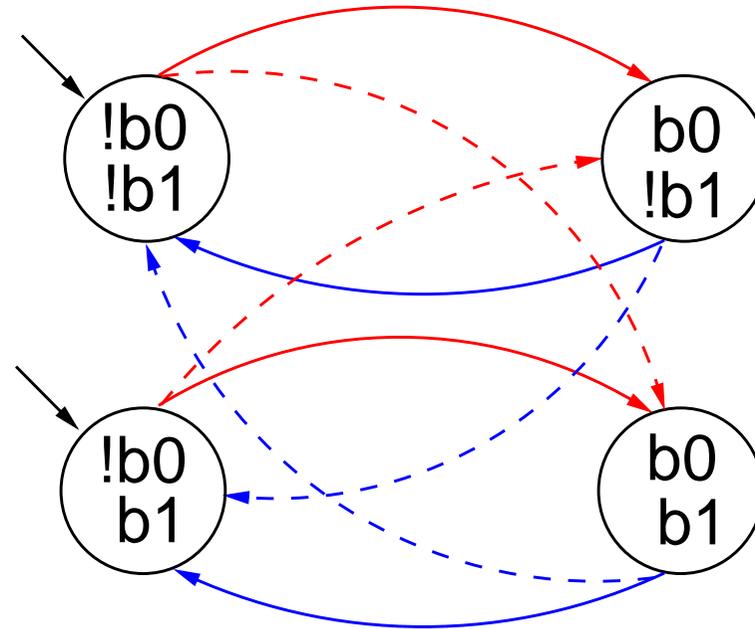
```
VAR
  n : 1..8;
```

Adding a state variable



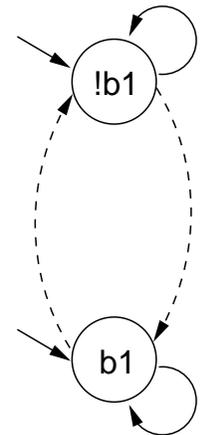
```
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

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.

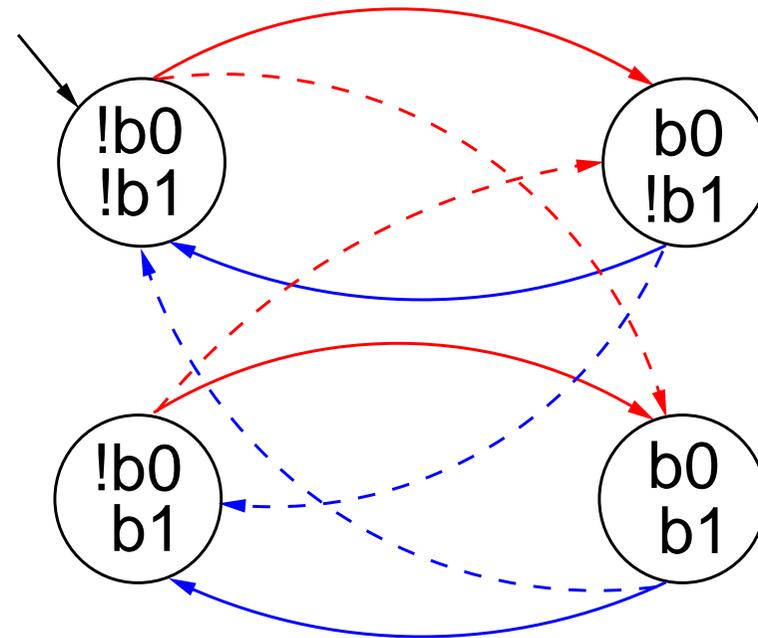
Declaring the set of initial states



```
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.

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

- ➔ 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;
```

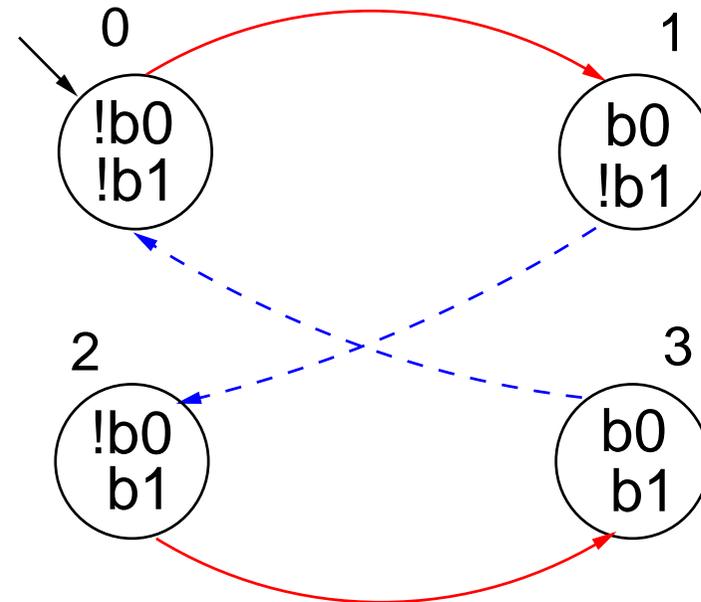
```
ASSIGN
```

```
  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
```

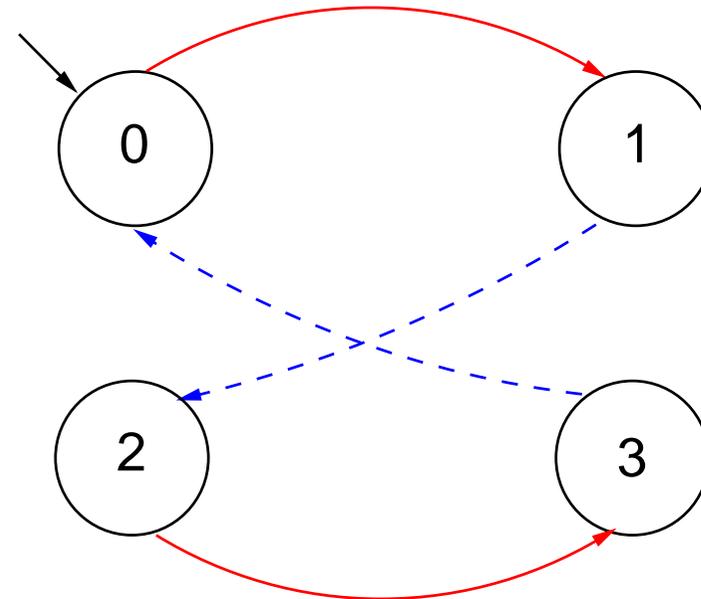
```
  init(b0) := 0;
```

```
  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
```

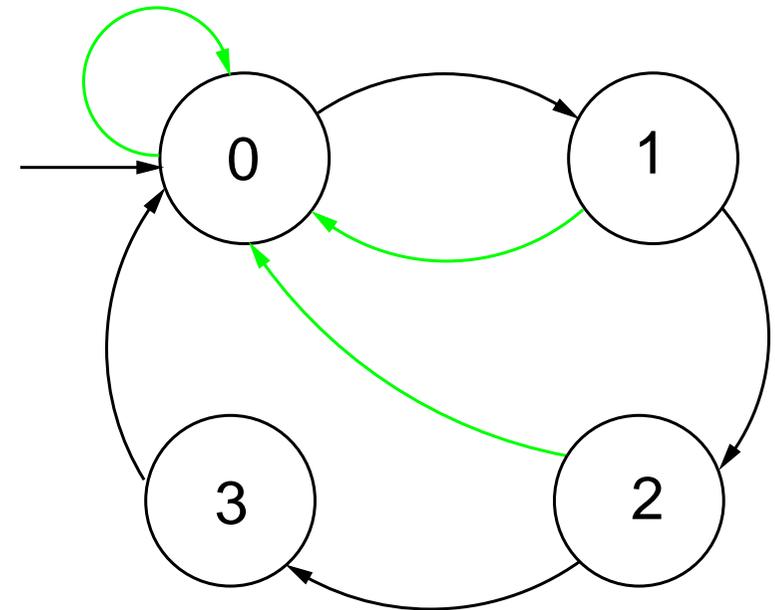
```
  b0      : boolean;  
  b1      : boolean;  
  reset   : boolean;  
  out     : 0..3;
```

```
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;
```



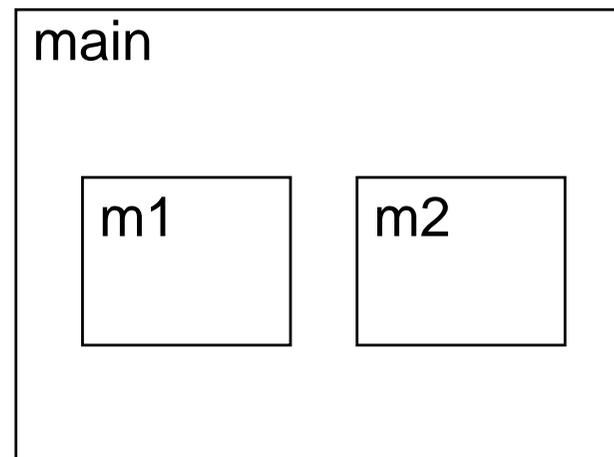
Modules



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;
```



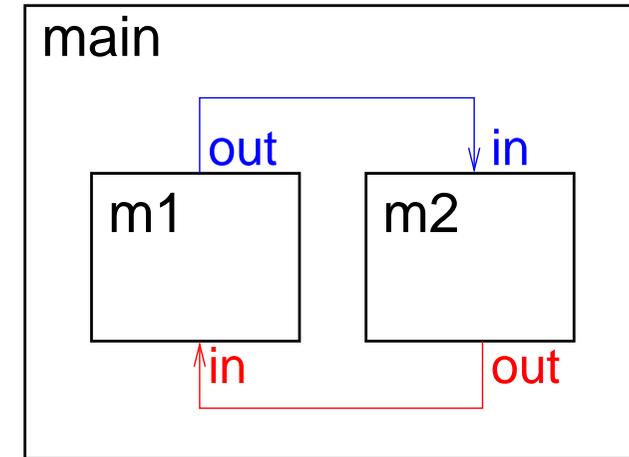
- ➡ Modules are instantiated in other modules. The instantiation is performed inside the `VAR` declaration of the parent module.
- ➡ In each SMV specification there must be a module `main`. It is the top-most module.
- ➡ All the variables declared in a module instance are visible in the module in which 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 $\text{out} = 3$ holds.

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

☞ A state in which $\text{out} = 3$ is reached in 2 steps.

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

☞ From any reachable state, a state in which $\text{out} = 3$ is reached in 3 steps.

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

Fairness Constraints



Let us consider again the counter with reset.

➡ The specification $\text{AF } \text{out} = 1$ 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 $\text{AF } \text{out} = 2$. 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

$$\text{FAIRNESS}$$
$$\text{out} = 1$$

we restrict our analysis to paths in which the property $\text{out} = 1$ is true infinitely often.

- ➔ The property $\text{AF } \text{out} = 1$ under this fairness constraint is now verified.
- ➔ The property $\text{AF } \text{out} = 2$ is still not verified.
- ➔ Adding the fairness constraint $\text{out} = 2$, then also the property $\text{AF } \text{out} = 2$ 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};`

`next(state) := ready;`

`out := b0 + 2*b1;`

INIT `state in {ready,busy}`

TRANS `next(state) = ready`

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
```

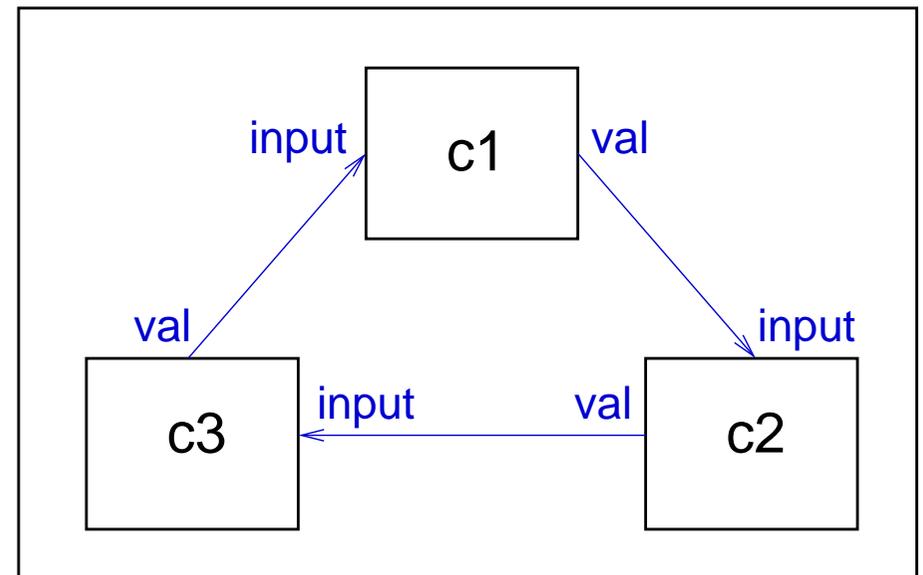
Synchronous composition



- ➔ 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:

<i>step</i>	<i>c1.val</i>	<i>c2.val</i>	<i>c3.val</i>
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:

<i>step</i>	<i>running</i>	<i>c1.val</i>	<i>c2.val</i>	<i>c3.val</i>
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/>