Model-Checking

Acknowledgment

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

The goals of the course are:

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

Authors of the slides:

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

Formal Verification

- Formal verification means to apply mathematical arguments to prove the correctness of systems
- Systems have bugs
  - Formal verification aims to find and correct such bugs

Why?

- Computer systems are getting more complex and pervasive, and bugs are unacceptable (mission control, medical devices) or prohibitively expensive (Pentium FDIV, Buffer overruns)
- In hardware, 70% of design effort goes into verification, with twice as many verification engineers than RTL designers
- In software, the numbers are similar

What kind of bugs?

- Concurrency errors
- Scenario: You are designing a
  - 100K gate ASIC: perhaps 100 concurrent modules
  - Flight control system: dozens of concurrent processes, on multiple CPUs
  - Networked embedded system: tens of thousands of motes
- Under test, the system fails once in three days
  - The error is not reproducible
  - You cannot collect enough real-time data to find the bug
- Concurrency Error
  - Events x and y occur concurrently (say) every 10^{10} cycles
  - The designer did not realize events x and y could interact concurrently

Concurrency Bugs

x := 0 init

x := x + 1 || x := x - 1

post x = 1 !

- This one is easy!
- This can be prevented by using semaphores (locks)
- Other bugs are not so simple
  - Routing loop in AODV implementations
  - Gigamax cache coherence protocol: required 13 messages in sequence

What kind of bugs?

- Sequential programs
- Scenario: Your OS kernel crashes with mangled memory state
- Under test, the system fails once in three days
  - The error is not reproducible
  - You cannot collect enough real-time data to find the bug
- Bug: The stack overflowed, and wrote parts of memory
- Bug: Certain data structure invariants were not met

What is formal verification?

- Build a mathematical model of the system:
  - what are possible behaviors?
- Write correctness requirements in a specification language:
  - what are desirable behaviors?
- Analysis: (Automatically) check that model satisfies specification
- Formal) Correctness claim is a precise mathematical statement
- Verification) Analysis either proves or disproves the correctness claim

Why study verification?

- General approach to improving reliability of systems
  - Hardware, systems software, embedded control systems, network protocols, networked embedded systems, …
- Increasing industrial interest
  - All major hardware companies employ in-house verification groups: Intel, Motorola, AMD, Lucent, IBM, Fujitsu, …
  - Tools from major EDA players: Synopsys Magellan, FormalCheck
  - Bunch of start-ups: Calypto, Jasper, 0-In
  - SLAM project at Microsoft http://research.microsoft.com/slam
  - Coverity

Where is Verification Used?

- Hardware verification
  - Success in verifying microprocessor designs, ISAs, cache coherence protocols
  - Fits in design flow
  - Tools: SMV, nuSMV, VIS, Mocha, FormalCheck
- Protocol verification
  - Network/Communications protocol implementations
  - Tools: Spin
- Software verification
  - Apply directly to source code (e.g., device drivers)
  - Tools: SLAM, Blast, Magic
- Embedded and real time systems
  - Tools: Uppaal, HyTech, Kronos, Charon

Formal Methods: Solution and Benefits

- The problem:
  - Certain (sub)systems can bee 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 advanced verification techniques (e.g. model checking).

FM 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 MC

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

Model Checking

- Model checking is an **automatic verification technique** for
- finite state concurrent systems.
  - • Developed independently by **Clarke and Emerson** and by **Queille and Sifakis** in early 1980's.
- • **Specifications** are written in **propositional temporal logic.**
- • Verification procedure is an **exhaustive search of the state space** of the design.

- Model Checking is a formal verification technique
  - analysis of complex reactive systems: hardware designs, communication protocols, embedded control systems for railways/avionics
- 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).
- Powerful debugging capabilities:
  - – detect costly problems in early developmemt stages (cfr. Pentium bug);
  - – exhaustive, thus effective (often bugs are also in scaled-down problems).
  - – provides counterexamples (directs the designer to the problem).

Model Checking in a nutshell

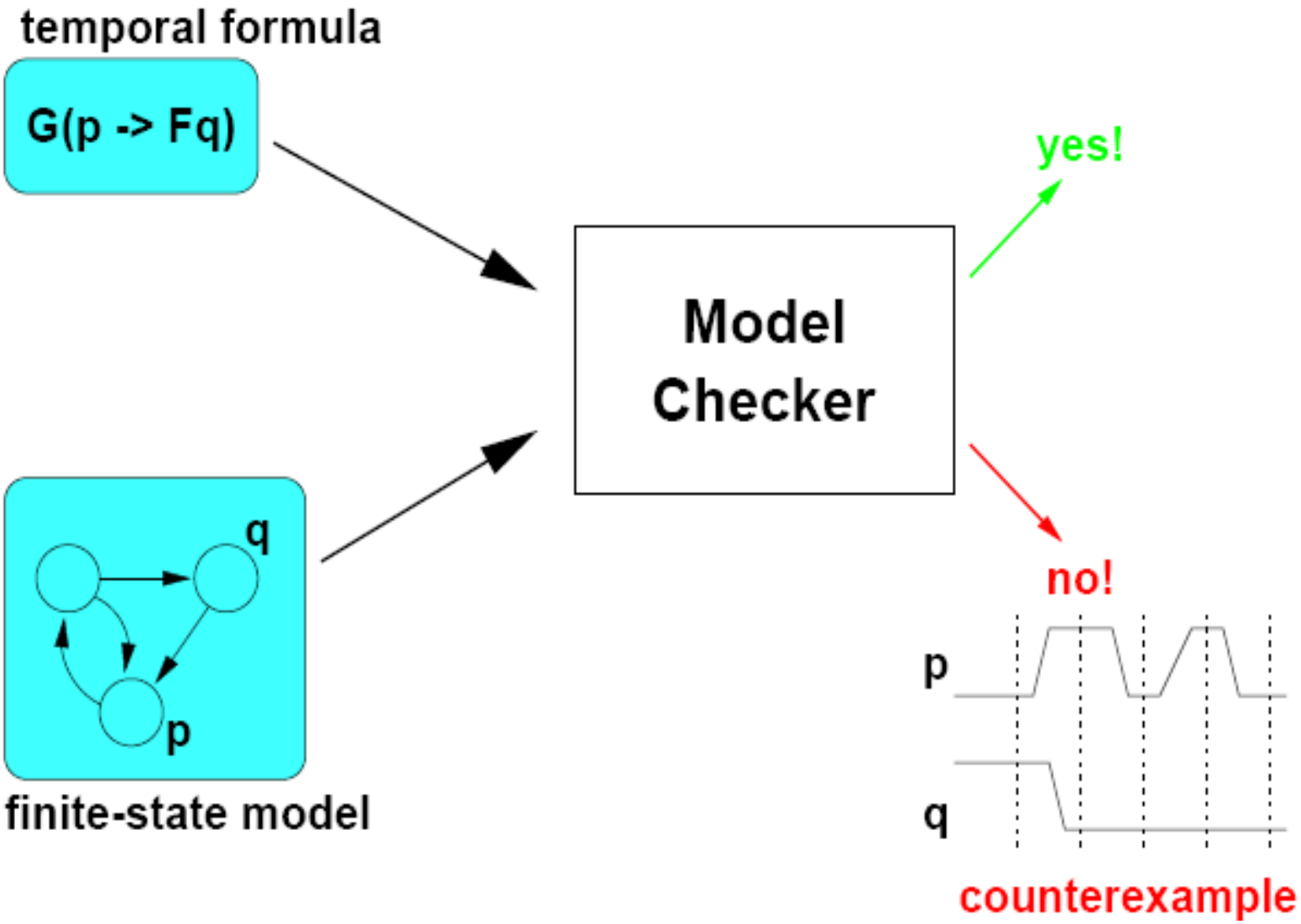
- Reactive systems represented as a finite state models
  - (in this course, Kripke models).
- System behaviors represented as (possibly) infinite sequences of states.
- Requirements represented as formulae in temporal logics.
- *"The system satisfies the requirement"* represented as truth of the formula 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.

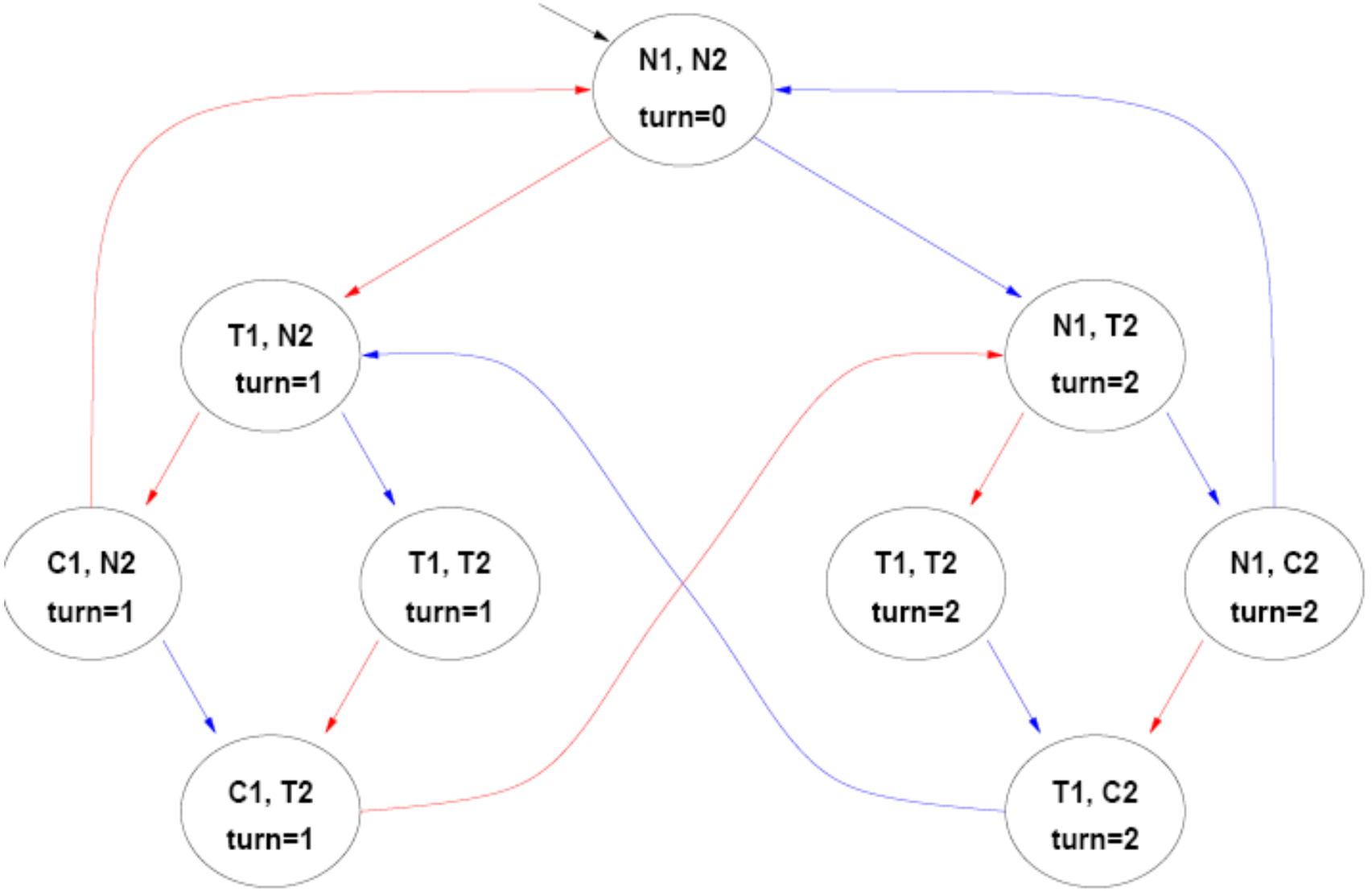
What is a Model Checker



We will not discuss

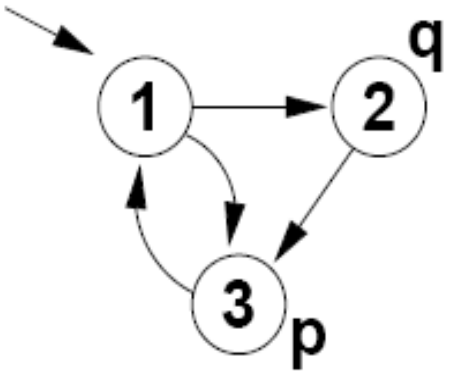
- A deep theoretical background. We will focus on practice.
- Advanced model checking techniques:
  - – abstraction;
  - – compositional, assume-guarantee reasoning;
  - – symmetry reduction;
  - – approximation techniques (e.g. directed to bug hunting);
  - – model transformation techniques (e.g. minimization wrt to bisimulation)

A Kripke model for mutual exclusion



Modeling the system: Kripke models

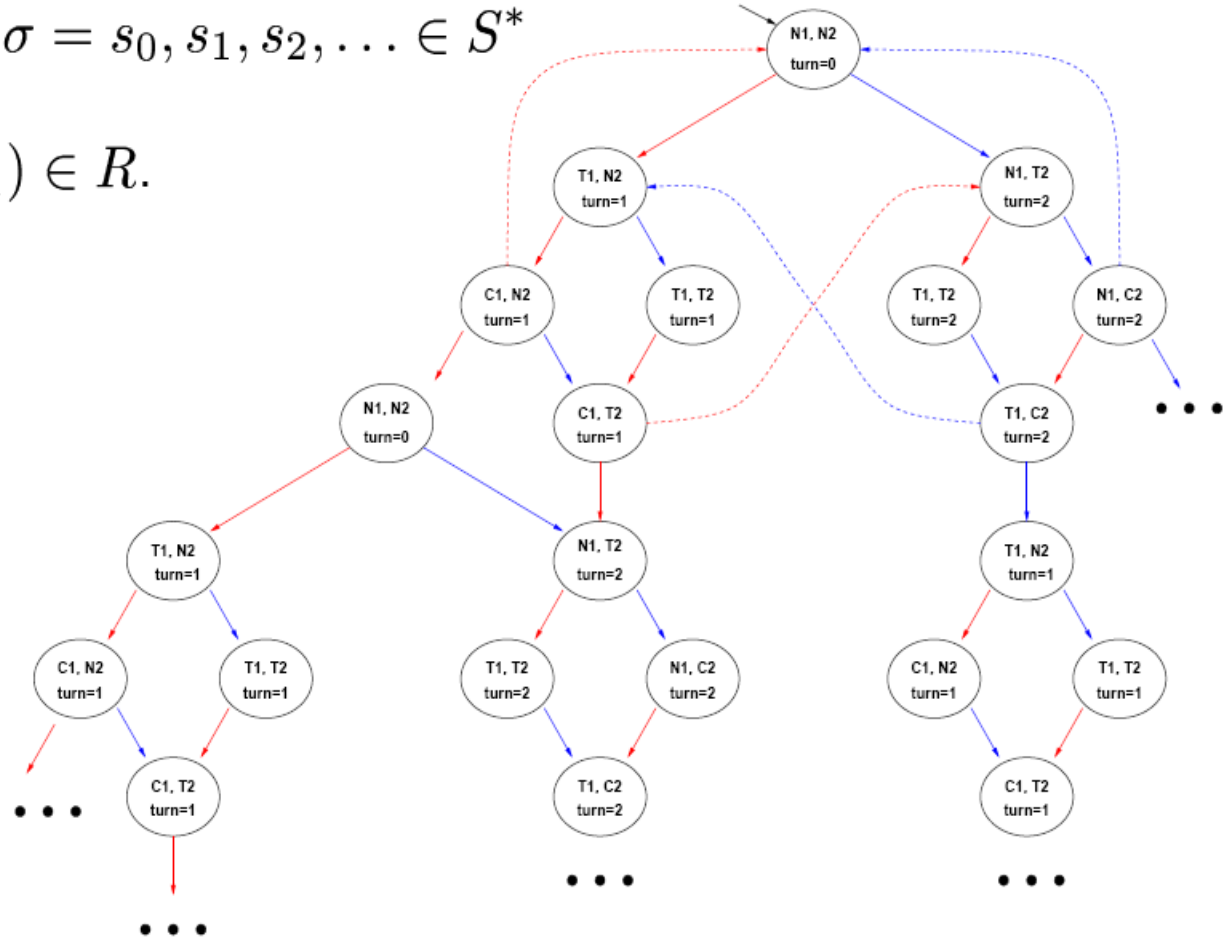
- Kripke models are used to describe reactive systems:
  - nonterminating systems with infinite behaviors,
  - e.g. communication protocols, operating systems, hardware circuits;
  - represent dynamic evolution of modeled systems;
  - values to state variables, program counters, content of communication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



A path in a Kripke model M is an infinite sequence

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

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



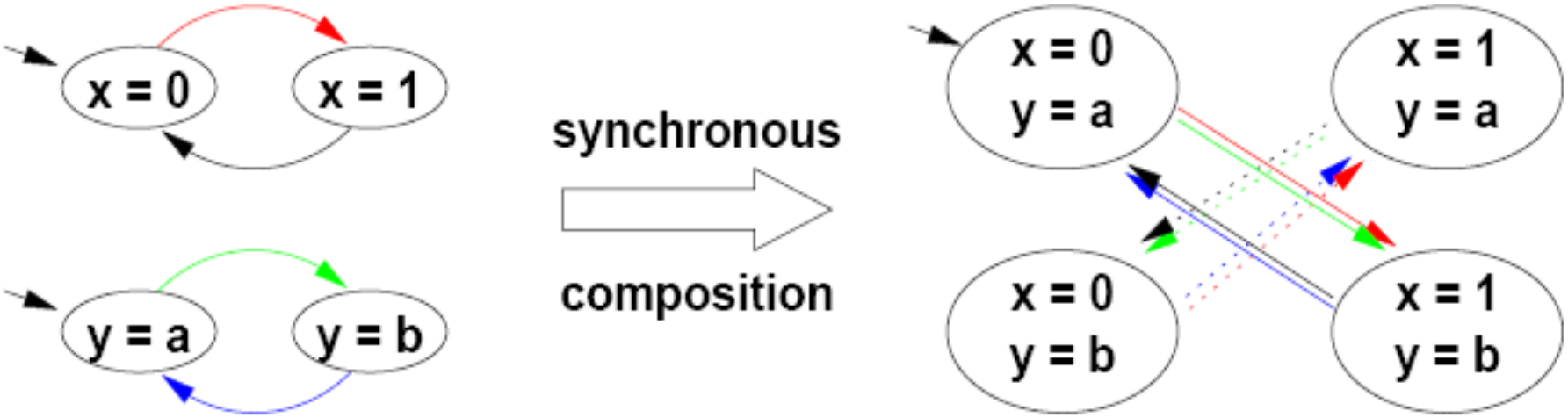
A state s is reachable in M if there is a path from the initial states to s.

Description languages for Kripke Model

- 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
  - instructions: determine the transition relation
- 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

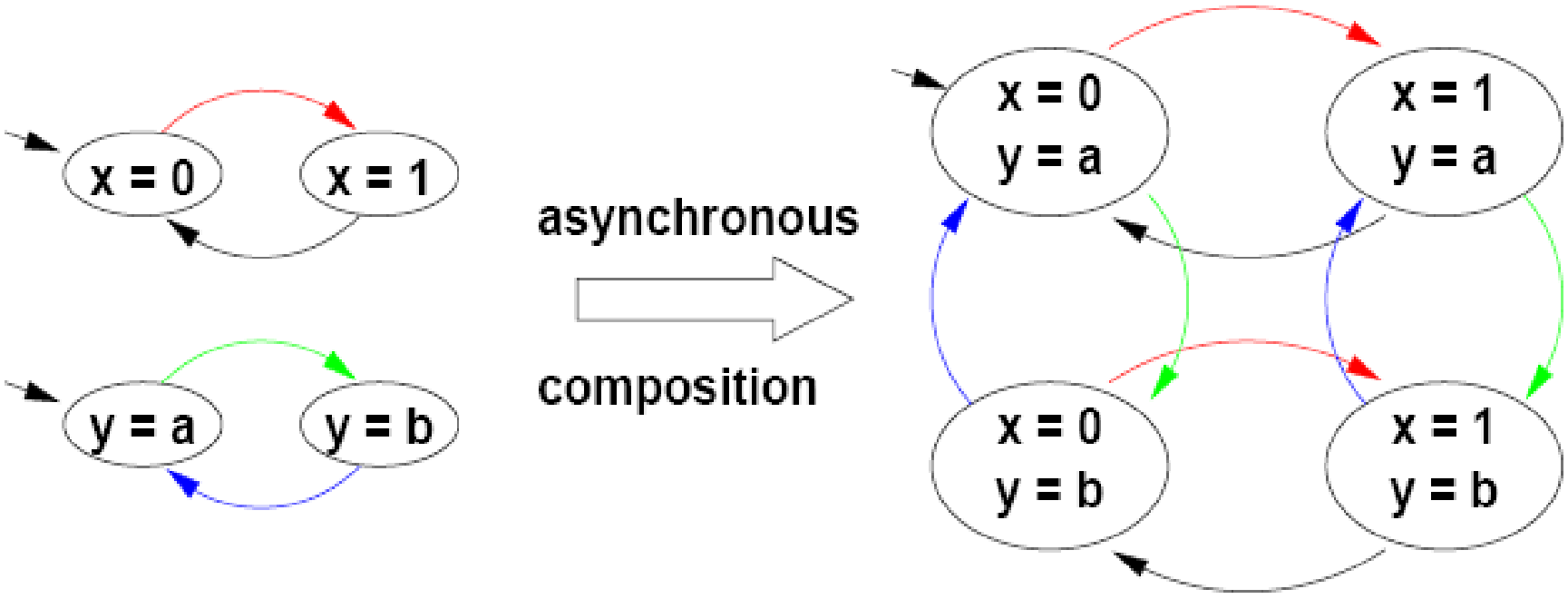
- Components evolve in parallel.
- At each time instant, every component performs a transition.



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

Async Composition

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

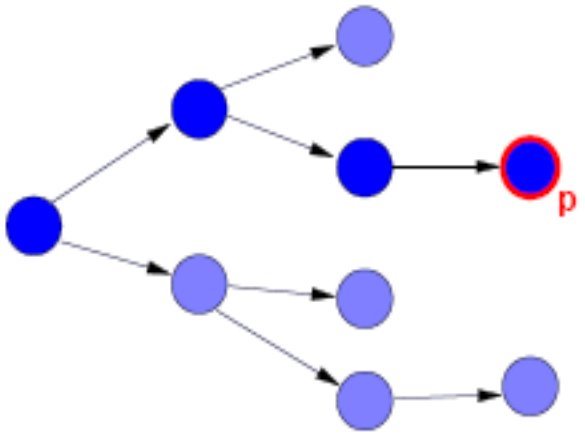


- Typical example: communication protocols.
- Asynchronous composition can be represented with NuSMV processes.

Properties

Safety properties:

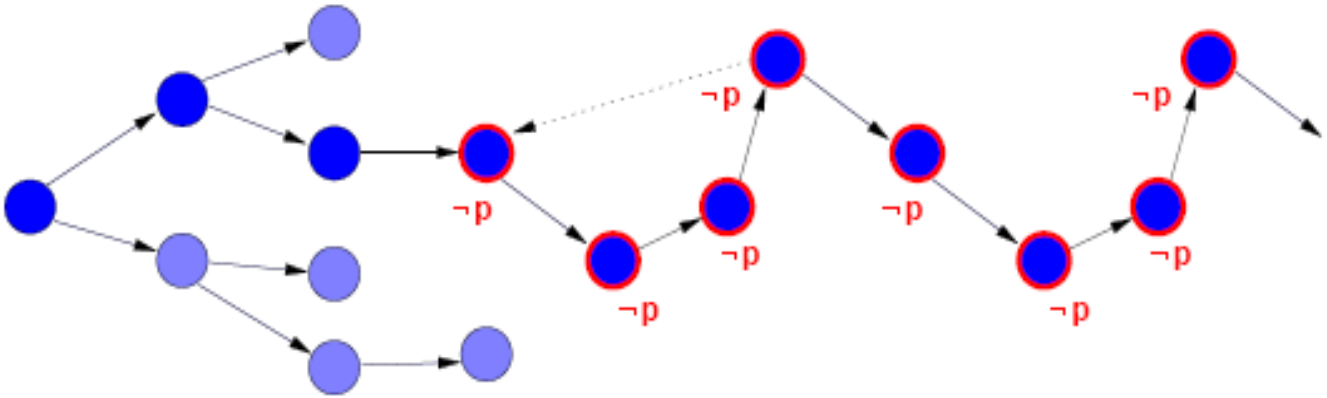
- nothing bad ever happens
  - deadlock: two processes waiting for input from each other, the system is unable to perform a transition.
  - no reachable state satisfies a “bad” condition, e.g. never two process in critical section at the same time
- can be refuted by a finite behaviour
- it is never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



Properties

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



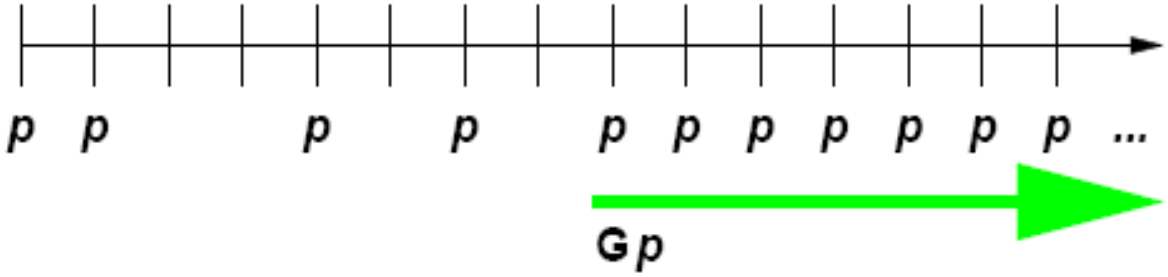
  - an infinite behaviour can be presented as a loop

Temporal Logics

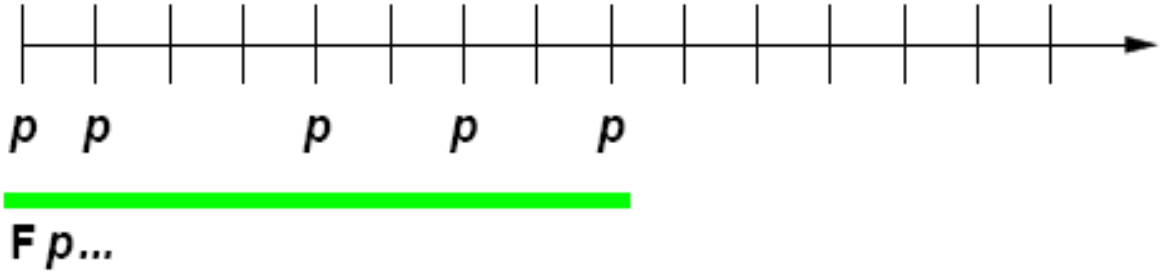
- Express properties of “Reactive Systems”
  - – nonterminating behaviours,
  - – without explicit reference to time.
- Linear Time Temporal Logic (LTL)
  - – intepreted over each path of the Kripke structure
  - – linear model of time
  - – temporal operators
- Computation Tree Logic (CTL)
  - – interpreted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

Temporal Operators

- "Globally": Gp at t iff p for **all** $t' \geq t$.

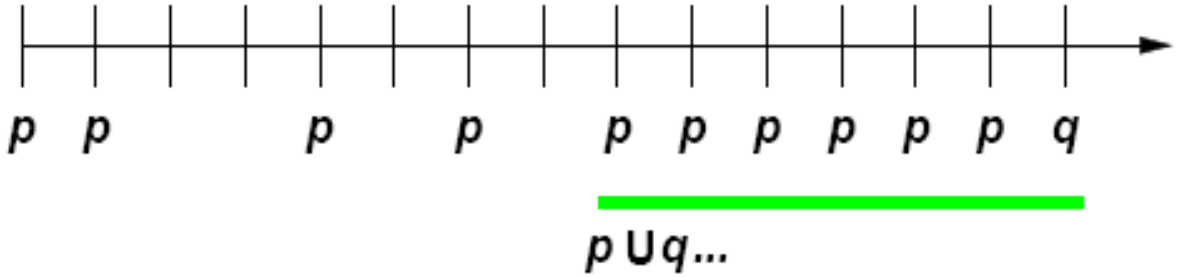


- "Future": Fp at t iff p for **some** $t' \geq t$.

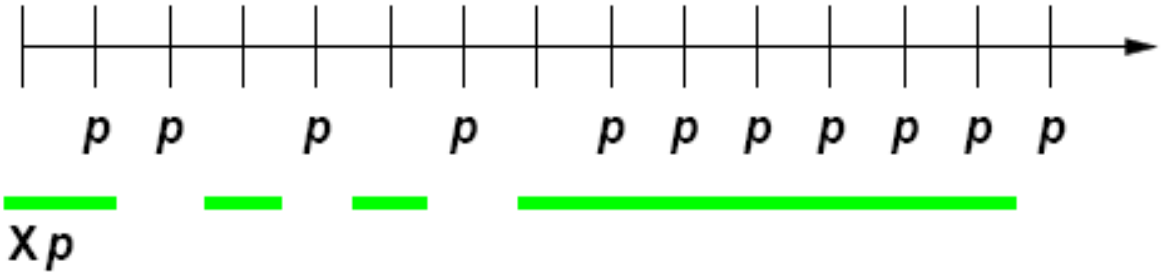


Temporal Operators

- "Until": pUq at t iff
  - q for some $t' \geq t$
  - p in the range $[t, t')$



- "Next-time": Xp at t iff p at $t+1$.



Examples

- Liveness: “if input, then eventually output”

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

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

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

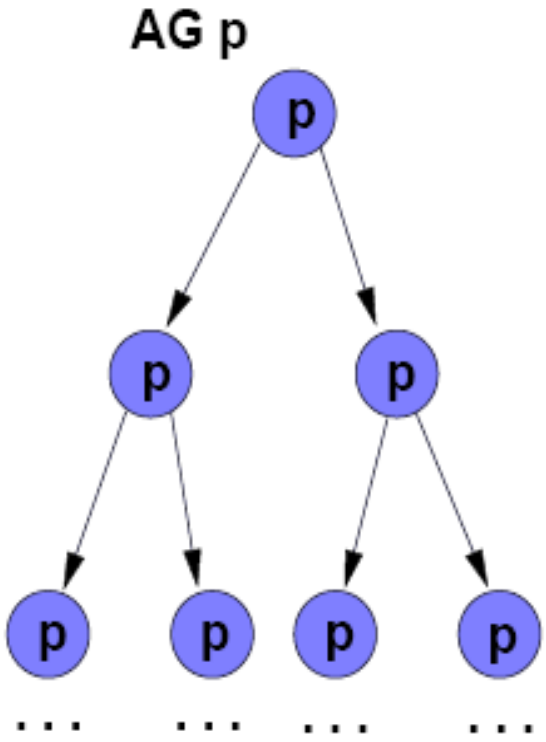
- Weak until: “no output before input”

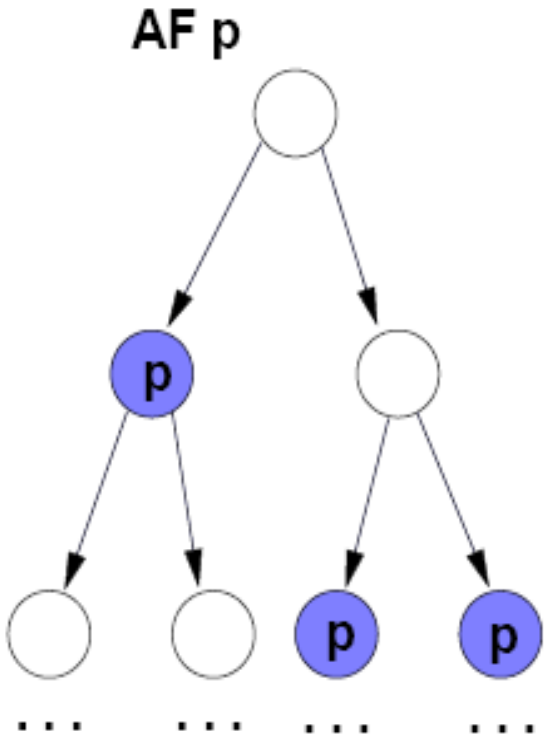
$$\neg\text{output}\ W\ \text{input}$$

where $p\ W\ q \quad \leftrightarrow \quad (p\ U\ q\ \vee\ Gp)$

Computational Tree Logic

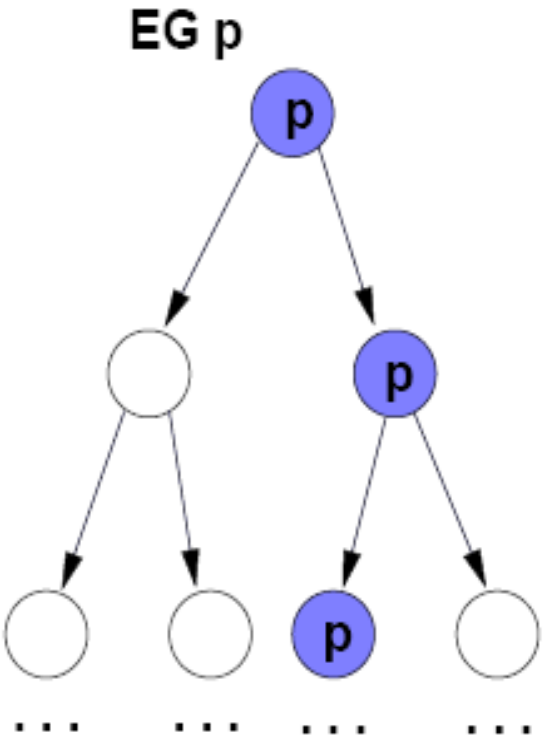
- Every temporal operator (F, G, X, U) preceded by a path quantifier (A or E).
- Universal modalities...

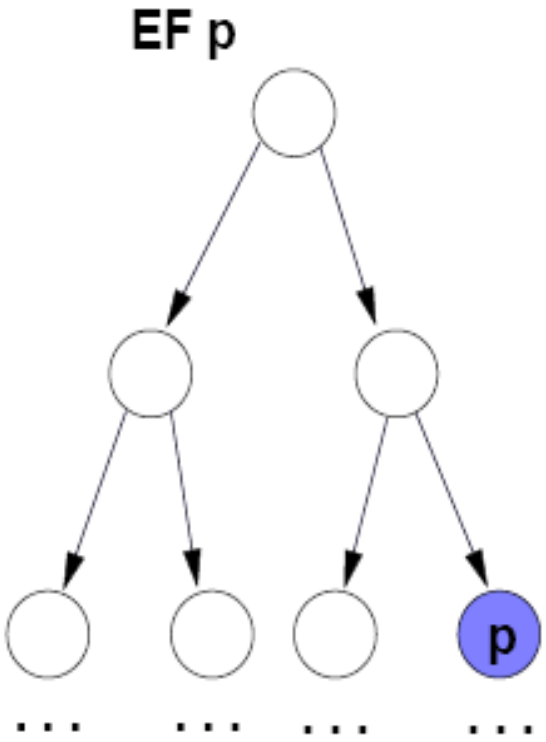




CTL

- Existential modalities...





CTL

- Other modalities:

$$AXp, EXp, A(pUq), E(pUq)$$

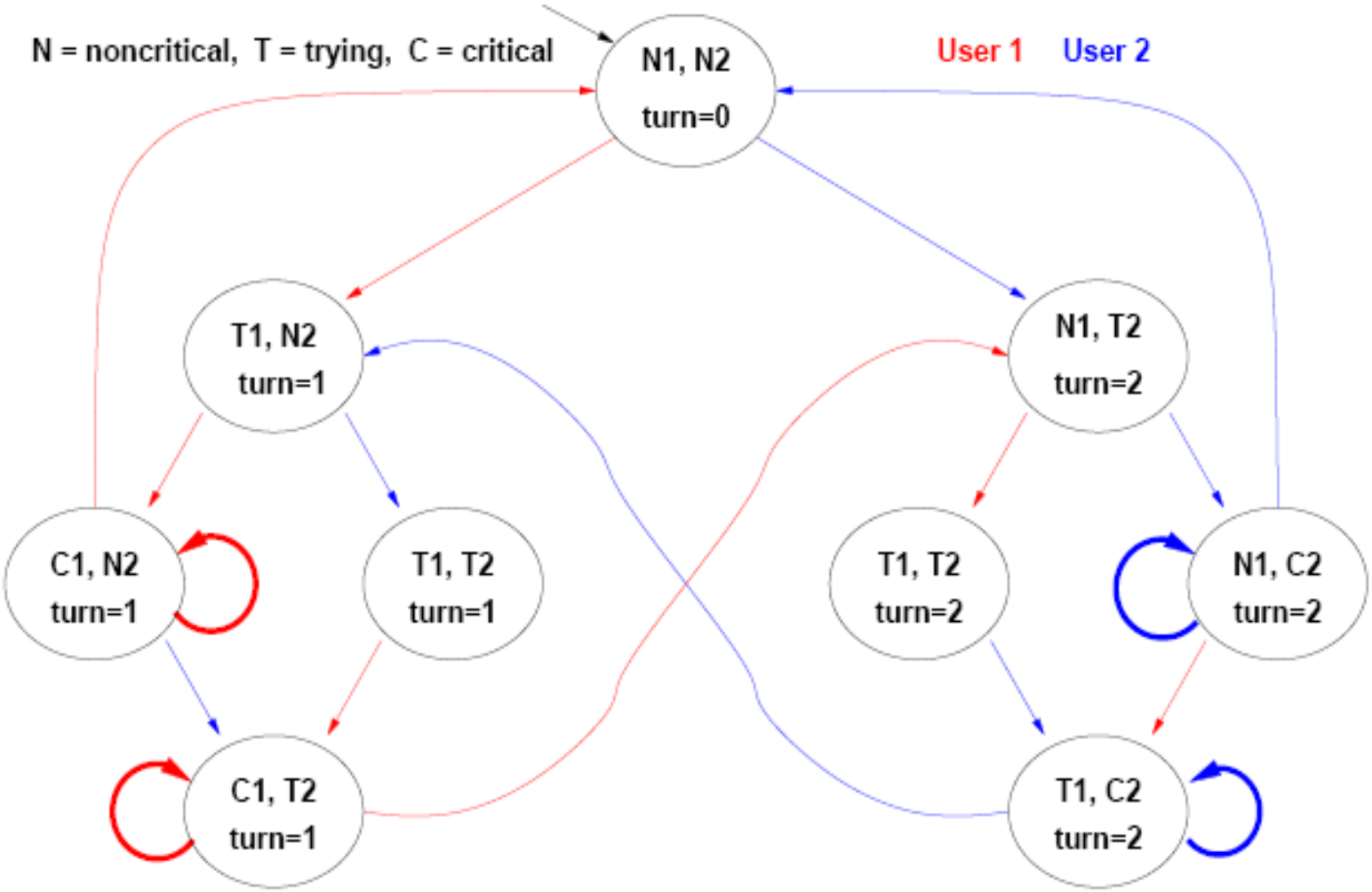
- Some dualities:

$$AGp \leftrightarrow \neg EF\neg p$$
$$AFp \leftrightarrow \neg EG\neg p$$

- Example: specifications for the mutual exclusion problem.

$AG\neg(C_1 \wedge C_2)$	mutual exclusion
$AG(T_1 \rightarrow AFC_1)$	liveness
$AG(N_1 \rightarrow EXT_1)$	non-blocking

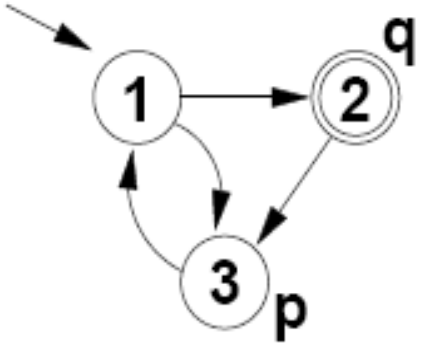
Need for Fairness



Fair Kripke Models

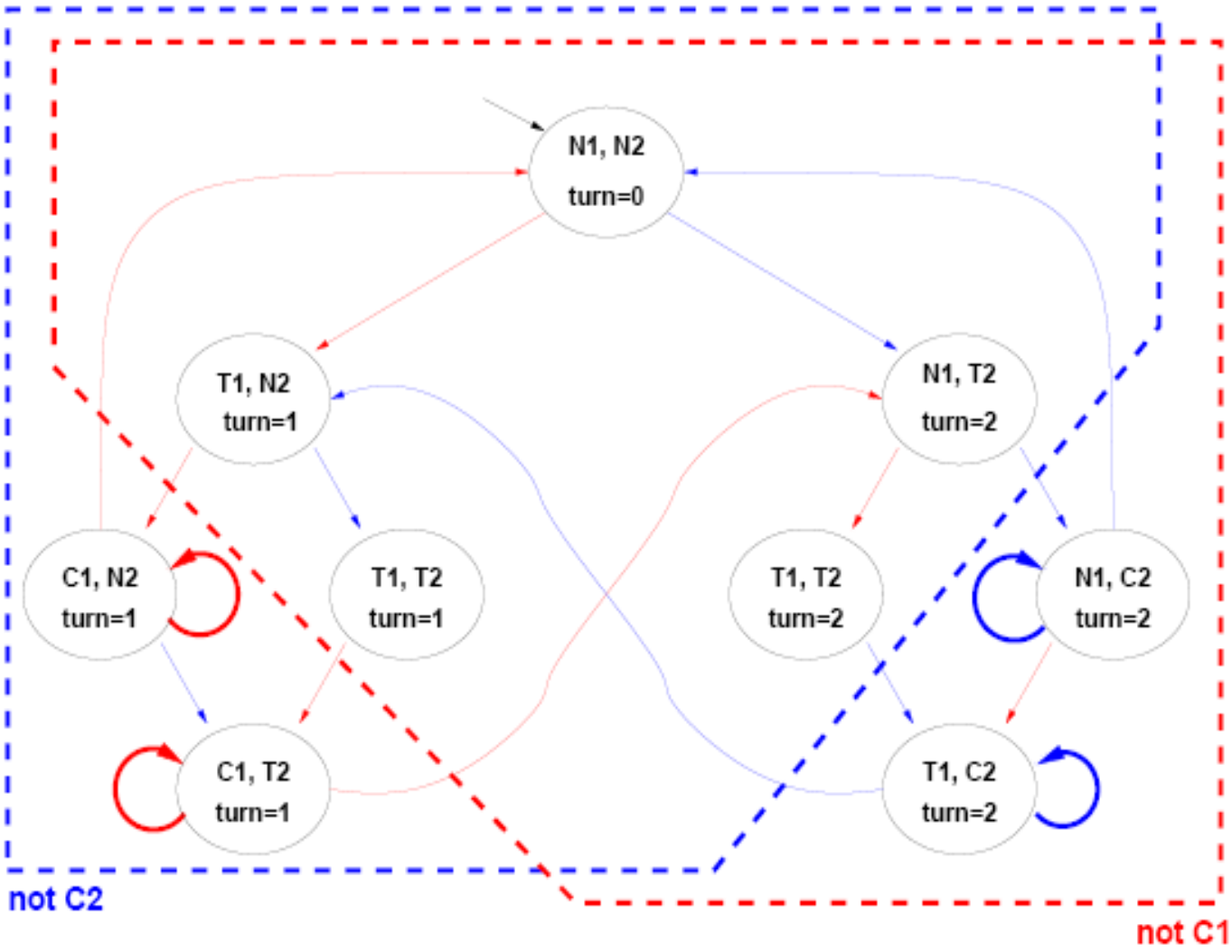
- Intuitively, fairness conditions are used to eliminate behaviours where a condition never holds
  - e.g. once a process is in critical section, it never exits
- Formally, a Kripke model (S, R, I, L, F) consists of
  - a set of states S;
  - a set of initial states $I \subseteq S$;
  - a set of transitions $R \subseteq S \times S$;
  - a labeling $L \subseteq S \times AP$.

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



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

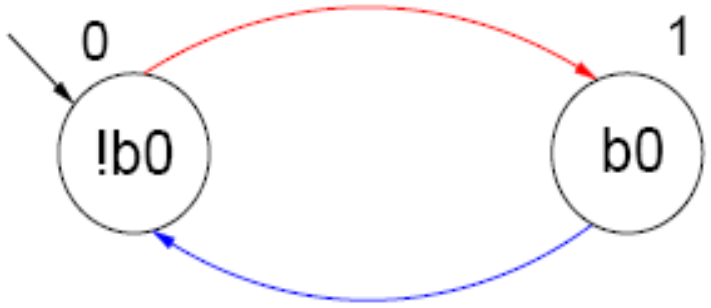
Fairness: {{ not C1},{not C2}}



NuSMV

The first SMV program

```
MODULE main
 VAR
   b0 : boolean;

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



An SMV program consists of:

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

Declaring State Variables

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

boolean:

```
VAR
  x : boolean;
```

enumerative:

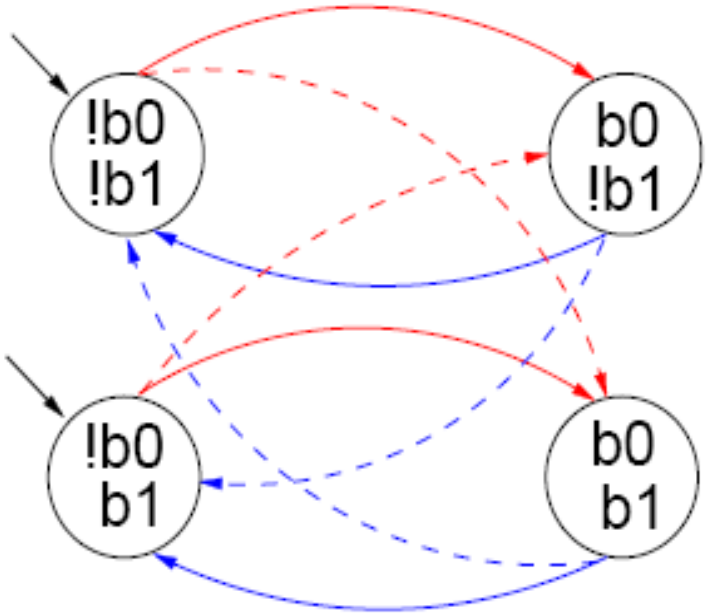
```
VAR
  st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

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

Adding a State Variable

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

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



Remarks:

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

Declaring the Set of Initial States

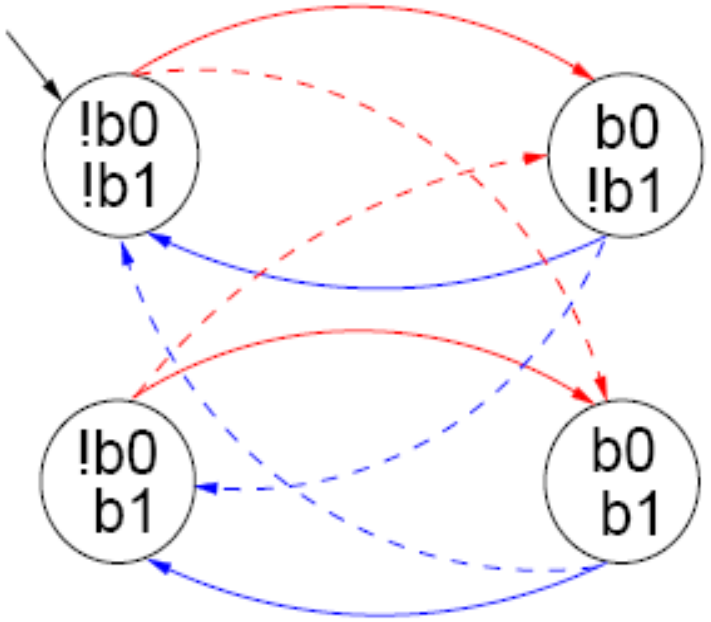
- For each variable, we constrain the values that it can assume in the *initial states*.

```
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.

Initial States

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

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

   init(b1) := 0;
```



Expressions

- Arithmetic operators:

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

- Comparison operators:

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

- Logic operators:

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

- Conditional expression:

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

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

- Set operators:

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

Expressions

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

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

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

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:

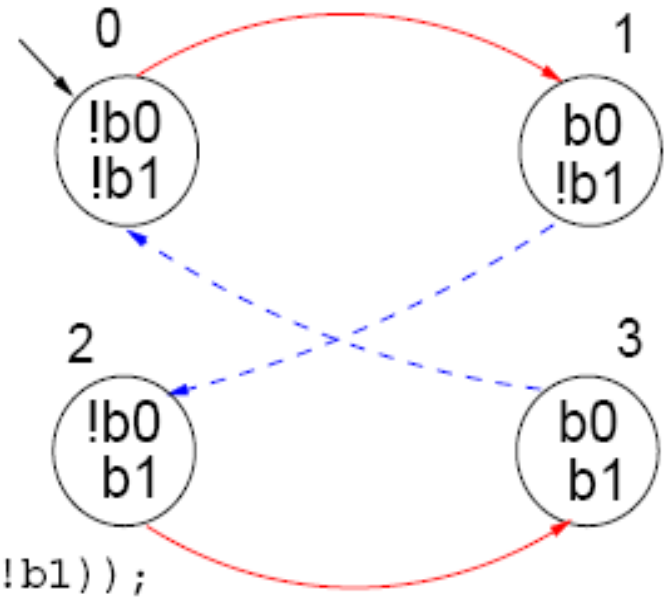
```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

- 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.

Transition

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

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

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



- (0,0)->(1, ((1&0)|(0&1)))=(1,0)
- (1,0)->(0, ((0&0)|(1&1)))=(0,1)

Normal Assignments

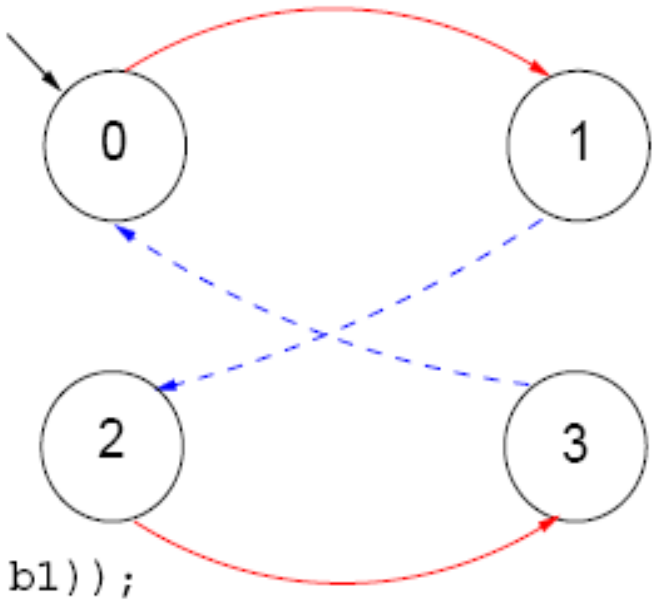
- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

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

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

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;
```



- (0,0,0)->(1,0,1+2*0)=(1,0,1)
- (1,0,1)->(0,1,0+2*1)=(0,1,2)

Restrictions on 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

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);
```

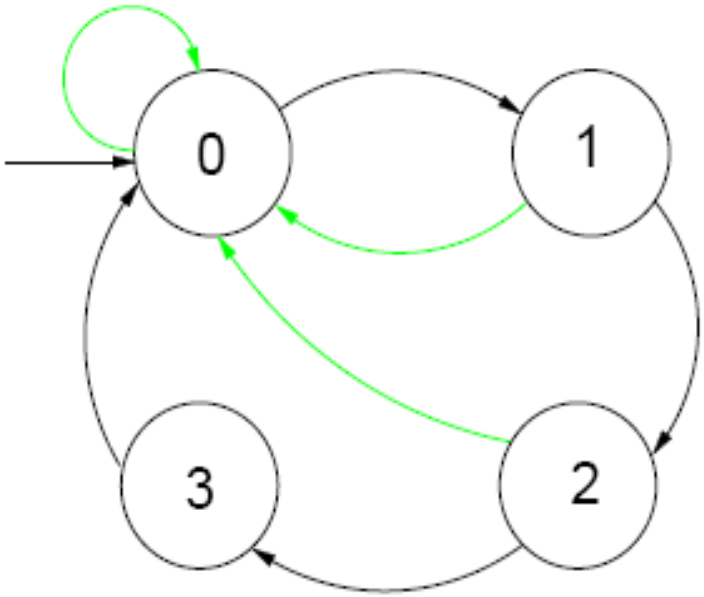
The following example is *legal*, instead:

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

Modulo 4 Counter w Reset

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

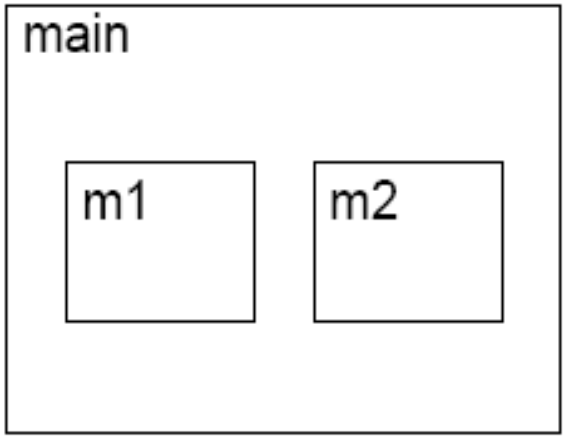
```
MODULE main
 VAR
   b0    : boolean;
   b1    : boolean;
   reset : boolean;
   out   : 0..3;

 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;
```

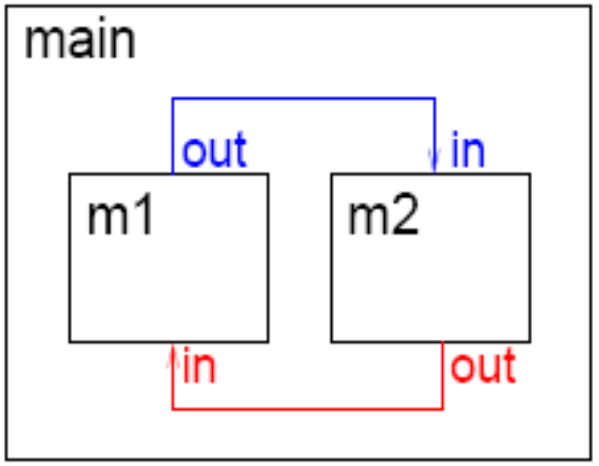


- Modules are instantiated in other modules. The instantiation is performed inside the `VAR` declaration of the parent module.
- In each SMV specification there must be a module `main`. It is the top-most module.
- All the variables declared in a module instance are visible in the module in which it 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.

Modulo 8 Counter

```
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`.

Module 8 Counter

```
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.

- See notes for sequence

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

- The SMV language allows for the specification of different kinds of properties:
  - invariants,
  - CTL formulas,
  - LTL formulas...
- Specifications can be added in any module of the program.
- Each specification is verified separately by NuSMV.

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)
```

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)
  ```

Fairness Constraints

Let us consider again the counter with reset.

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

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

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

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

  generates the loop:

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

  which is a counterexample to the given formula.

Fairness Constraints

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

```
FAIRNESS <simple_expression>
```

Fairness Constraints

- With the fairness constraint

```
FAIRNESS
  out = 1
```

  we restrict our analysis to paths in which the property `out = 1` is true infinitely often.

- The property `AF out = 1` under this fairness constraint is now verified.
- The property `AF out = 2` is still not verified.
- Adding the fairness constraint `out = 2`, then also the property `AF out = 2` is verified.

DEFINE

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;
```

- See notes for sequence

DEFINE

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;
```

DEFINE

- 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.

ASSIGN and DEFINE

- `VAR a: boolean;`
  `ASSIGN a := b | c;`
  - declares a new state variable `a`
  - becomes part of invariant relation

- `DEFINE d := b | c;`
  - is effectively a macro definition, each occurrence of `d` is replaced by `b | c`
  - no extra BDD variable is generated for `d`
  - the BDD for `b | c` becomes part of each expression using `d`

Arrays

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

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

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

Records

Records can be defined as modules without parameters and assignments.

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

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

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

Constraint Style

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
```

Constraint Style

- 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 vs. 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 vs. Constraints

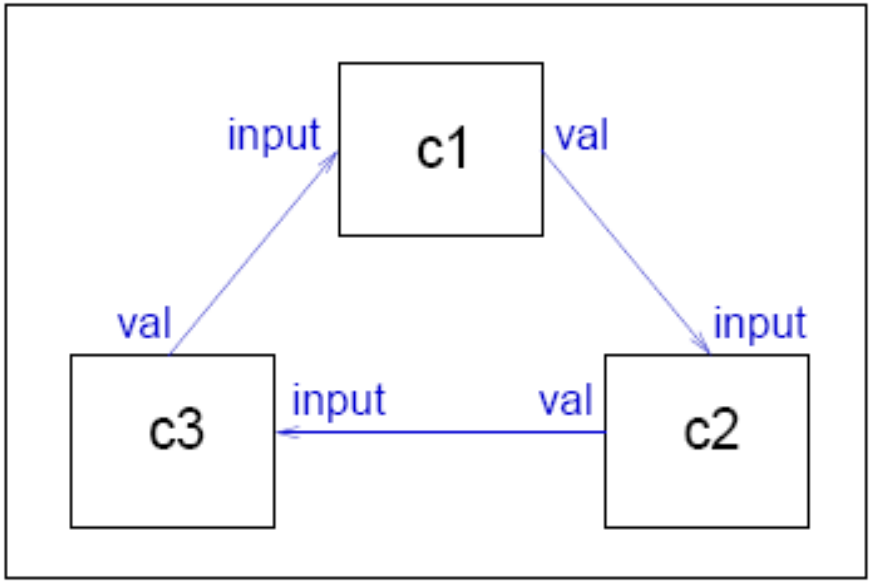
- Models written in **assignment style**:
  - by construction, there is always *at least one initial state*;
  - by construction, all states have *at least one next state*;
  - *non-determinism is apparent* (unassigned variables, set assignments...).
- Models written in **constraint style**:
  - `INIT`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

Sync 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);
```



Sync Composition

A possible execution:

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

Async 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);
```

A Possible Execution

A possible execution:

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

SMV Steps

- Read_Model : read model from input smv file
- Flatten_hierarchy : instantiate modules and processes
- Build_model : compile the model into BDDs (initial state, invar, transition relation)
- Check_spec : checking specification bottom up

Run SMV

- smv [options] inputfile
  - **-c cache-size for BDD operations**
  - **-k key-table-size for BDD nodes**
  - **-v verbose**
  - **-int interactive mode**
  - **-r**
    - prints out statistics about reachable state space

SMV Options

- –f
  - computes set of reachable states first
  - Model checking algorithm traverses only the set of reachable states instead of complete state space.
  - useful if reachable state space is a small fraction of total state space

SMV Options: Reordering vars

- Variable reordering is crucial for small BDD sizes and speed.
- Generally, variables which are related need to be close in the ordering.
- –i filename –o filename
  - Input, output BDD variable ordering to given file.
- -reorder
  - Invokes automatic variable reordering

SMV Options: Transition relation

smv -cp part_limit

- Conjunctive Partitioning: Transition relation not evaluated as a whole, instead individual next() assignments are grouped into partitions that do not exceed part_limit
- Uses less memory and benefits from early quantification

SMV options: -inc

- Perform incremental evaluation of the transition relation
- At each step in forward search, transition relation restriced to reached state set
- Cuts down on size of transition relation with overhead of extra computation

Example: Client & Server

```
MODULE client (ack)
VAR
  state : {idle, requesting};
  req : boolean;

ASSIGN
  init(state) := idle;
  next(state) :=
   case
    state=idle : {idle, requesting};
    state=requesting & ack : {idle, requesting};
    1 : state;
   esac;

  req := (state=requesting);
```

MODULE server (req)

```
VAR
  state : {idle, pending, acking};
  ack : boolean;

ASSIGN
  next(state) :=
   case
    state=idle & req : pending;
    state=pending : {pending, acking};
    state=acking & req : pending;
    state=acking & !req : idle;
    1 : state;
   esac;

  ack := (state = acking);
```

Is the specification true?

```
MODULE main
VAR
  c : client(s.ack);
  s : server(c.req);

SPEC AG (c.req -> AF s.ack)
```

- Need fairness constraint:
  - Suggestion:
    FAIRNESS s.ack
  - Why is this bad?
  - Solution:
    FAIRNESS (c.req -> s.ack)

NuSMV

- Specifications expressible in CTL, LTL and Real time CTL logics
- Provides both BDD and SAT based model checking.
- Uses a number of heuristics for achieving efficiency and control state explosion
- Higher number of features in interactive mode

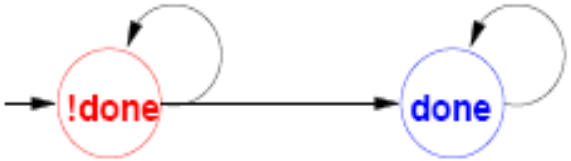
Cadence SMV

- Provides “compositional techniques” to verify large complex systems by decomposition to smaller problems.
- Provides a variety of techniques for refinement verification, symmetry reductions, uninterpreted functions, data type reductions.

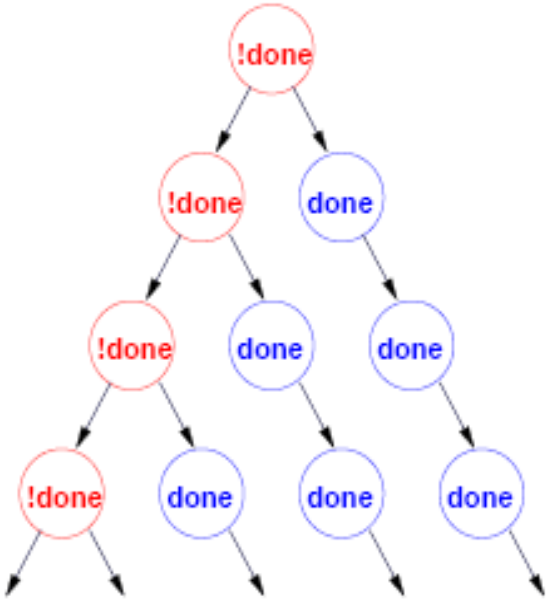
Paths and Trees

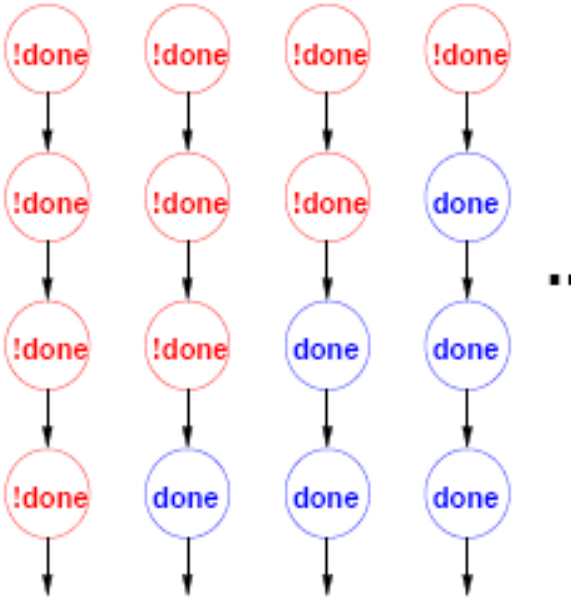
- An SMV specification defines a Kripke structure:

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



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





Specifications

In the SMV language:

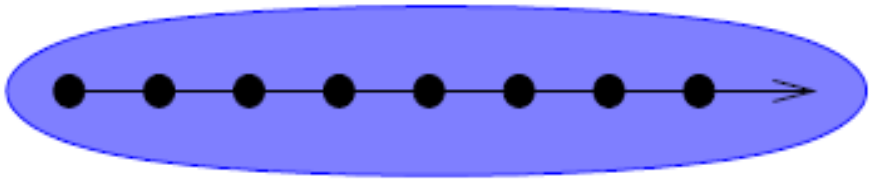
- Specifications can be added in any module of the program.
- Each property is verified separately.
- Different kinds of properties are allowed:
  - Properties on the reachable states
    - *invariants* (`INVARSPEC`)
  - Properties on the computation paths (*linear time* logics):
    - LTL (`LTLSPEC`)
    - qualitative characteristics of models (`COMPUTE`)
  - Properties on the computation tree (*branching time* logics):
    - CTL (`SPEC`)
    - Real-time CTL (`SPEC`)

LTL Specs

finally p

F p

globally p

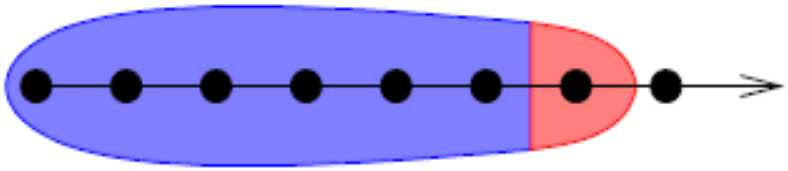


G p

next p

X p

p until q



p U q

LTL Specs

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

```
LTLSPEC <ltl_expression>
```

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

```
LTLSPEC F out = 3
```

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

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

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

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

Quantitative Properties

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

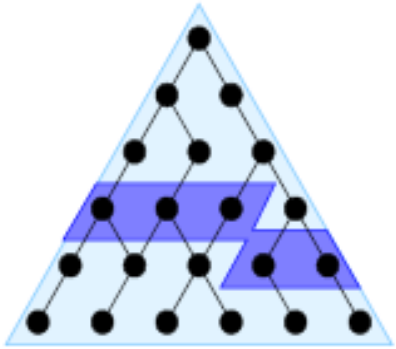
```
COMPUTE
  MIN [ out = 0 , out = 3]
```

- 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]
```

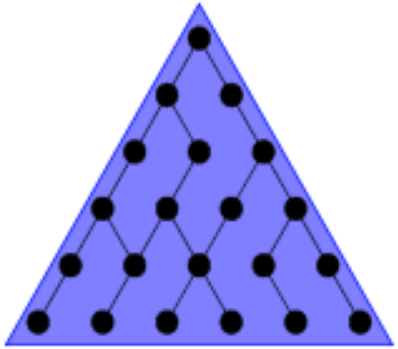
CTL Specs

finally p



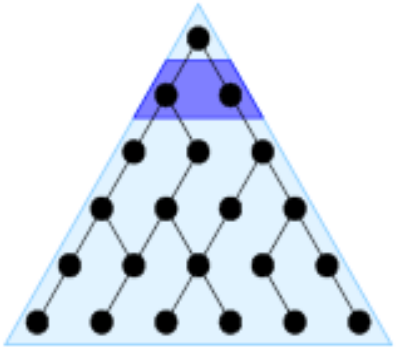
AF p

globally p



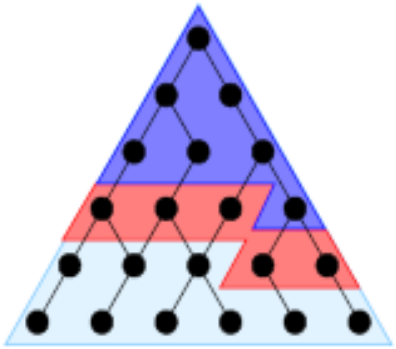
AG p

next p

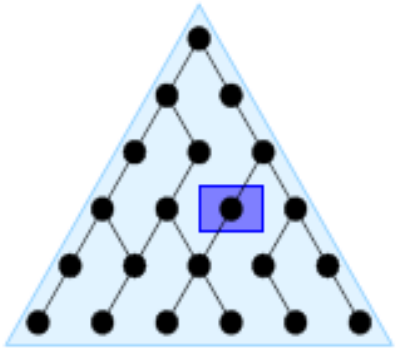


AX p

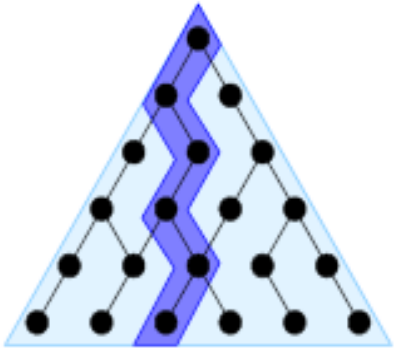
p until q



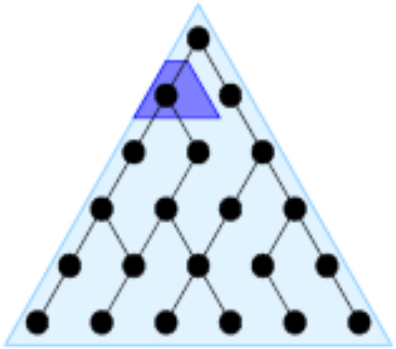
A [p U q]



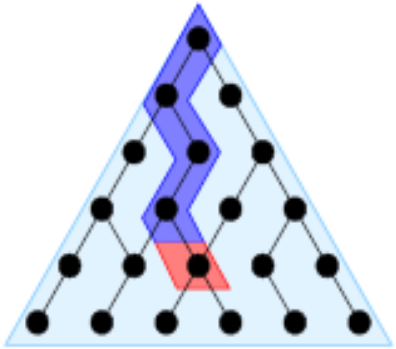
EF p



EG p



EX p



E [p U q]

CTL Specs

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

```
SPEC <ctl_expression>
```

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

```
SPEC EF out = 3
```

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

```
SPEC AF out = 3
```

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

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

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

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

Bounded CTL Specs

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

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

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

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

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

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

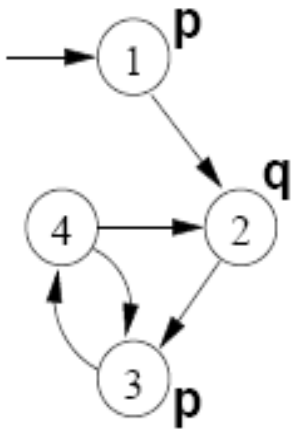
```
SPEC
  AG ABF 0..3 out = 3
```

Model-Checking Algorithms

Model-Checking

Model Checking is a formal verification technique where...

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



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

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

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

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:

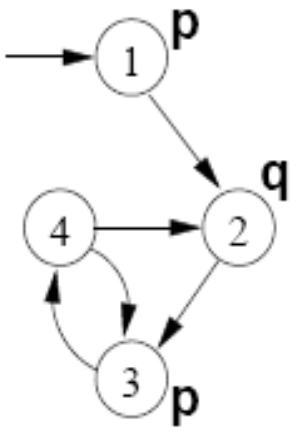
- manipulation of *sets of states* (rather than single states);
- sets of states represented by formulae in propositional logic;
  - set cardinality not directly correlated to size
- expansion of *sets of transitions* (rather than single transitions);
- two main symbolic techniques:
  - Binary Decision 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 MC Example

Consider a simple system and a specification:

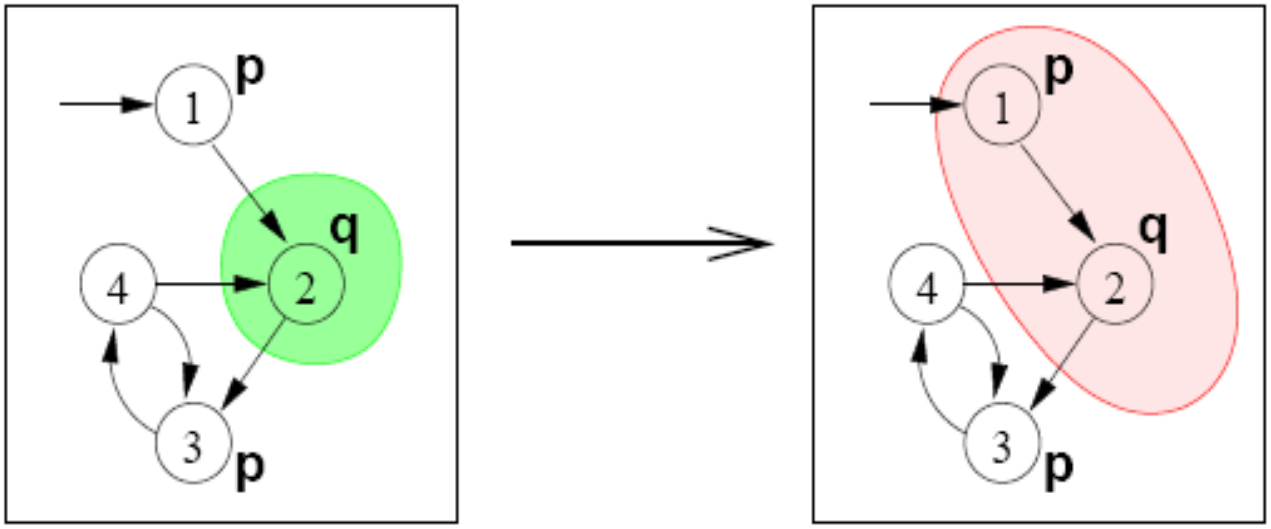


AG(p -> AFq)

Idea:

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

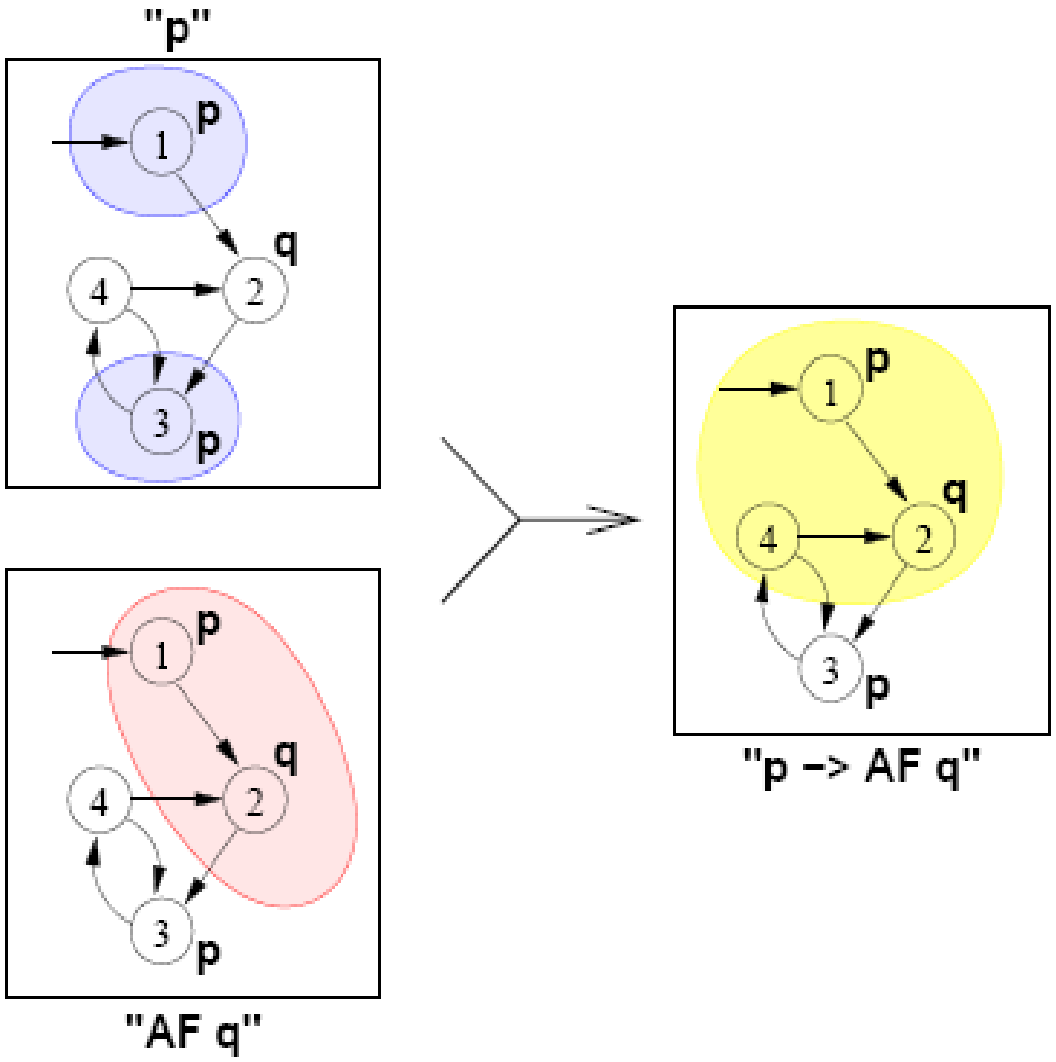
CTL MC Example



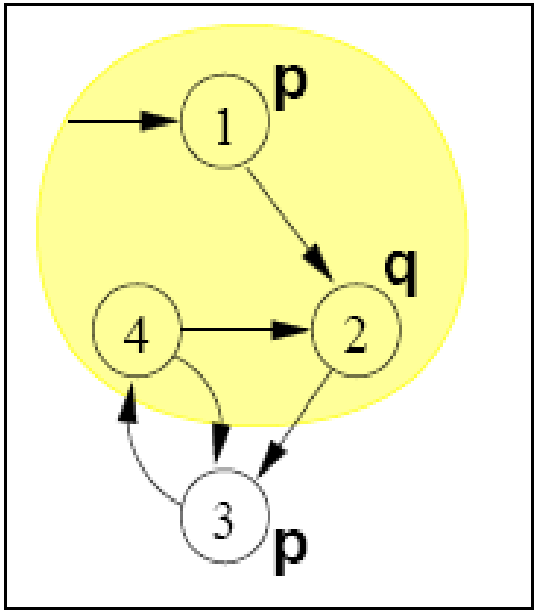
"q" "AF q"

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

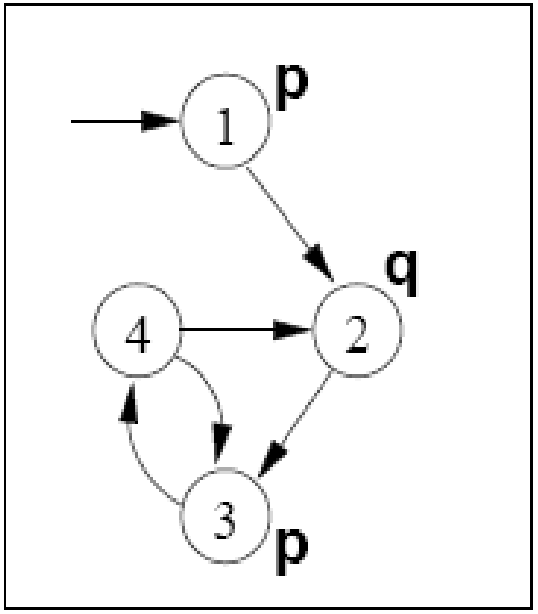
CTL MC Example



CTL MC Example



"p -> AF q"



"AG(p -> AF q)"

The set of states where the formula holds is empty!

Counterexample reconstruction is based on the intermediate sets.

Fixed Point SMC

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

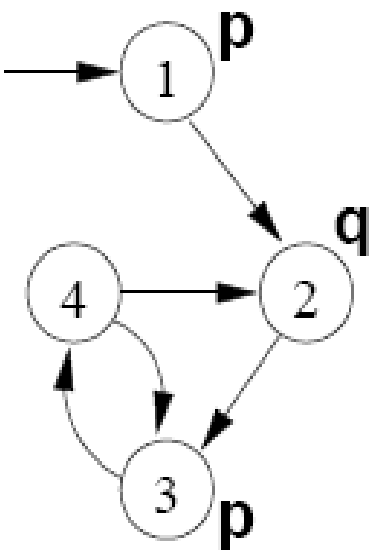
- traverse formula structure, for each subformula build set of satisfying states; compare result with initial set of states.
- boolean connectives: apply corresponding boolean operation;
- on $\mathrm{AX}\,\Phi$, apply preimage computation
  - $\forall \mathbf{s}'.(\mathcal{T}(\mathbf{s},\mathbf{s}')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

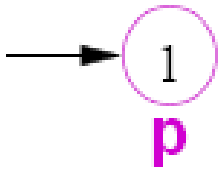
Bounded MC

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 $k \cdot |\mathbf{s}|$ 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)

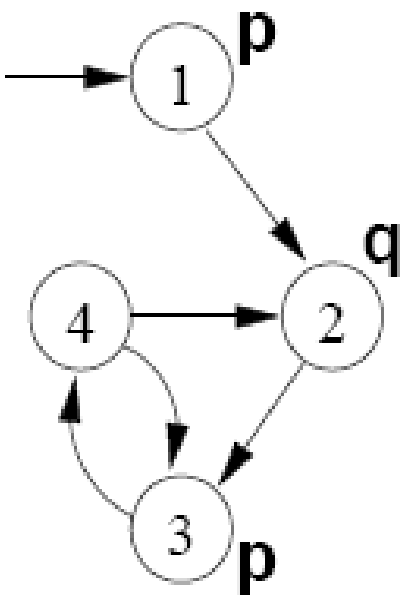
BMC Example



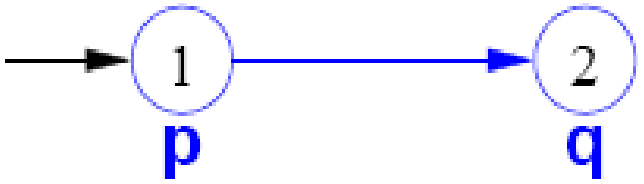
- Formula: **G(p -> Fq)**
- Negated Formula (violation): **F(p & G ! q)**
- $k = 0$: 

- No counter-example found.

BMC Example

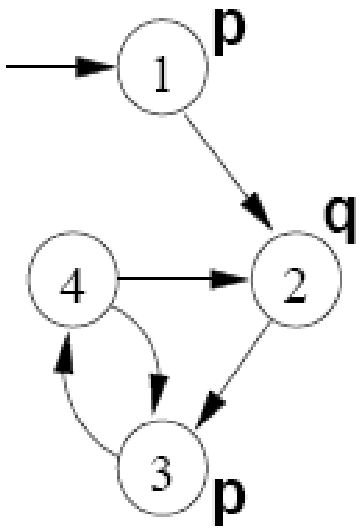


- Formula: **G(p -> Fq)**
- $k = 1$:

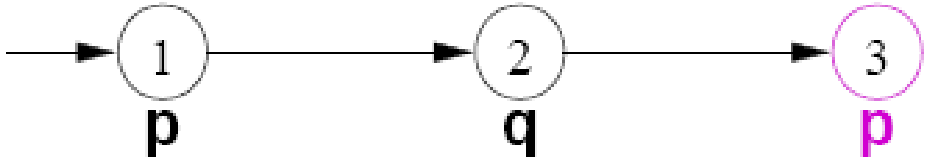


- No counter-example found.

BMC Example

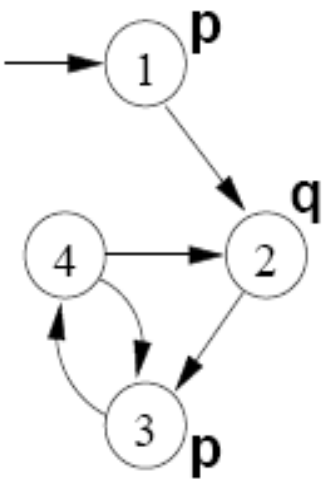


- Formula: **G(p –> Fq)**
- $k = 2$:

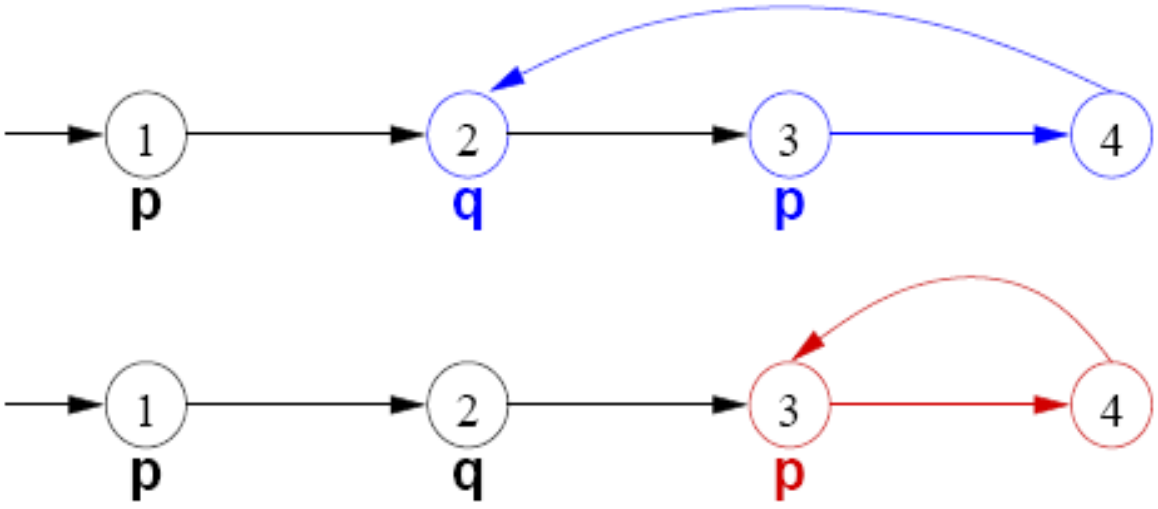


- No counter-example found.

BMC Example



- Formula: **G(p –> Fq)**
- $k = 3$:



- The 2nd trace is a counter-example!

BMC

- *Bounded Model Checking*:
  Given a FSM $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{T} \rangle$, 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 $\mathcal{T}$:

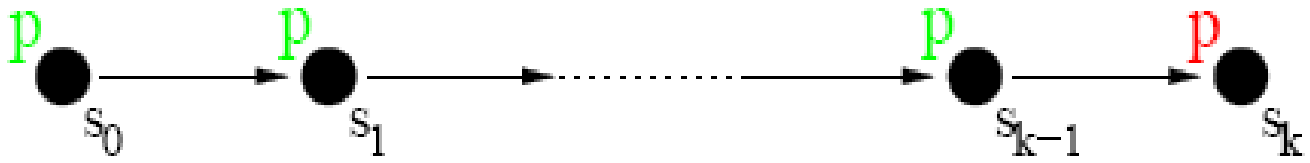
$$\mathcal{I}(\mathbf{s}_0) \wedge \mathcal{T}(\mathbf{s}_0, \mathbf{s}_1) \wedge \ldots \mathcal{T}(\mathbf{s}_{k-1}, \mathbf{s}_k)$$

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

BMC Examples

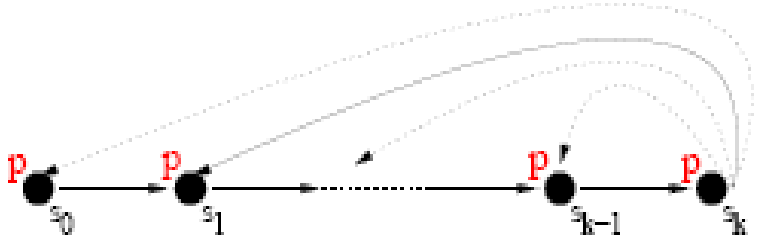
- $\phi = \mathrm{F}\, p$

$$[\![\mathrm{F}\, p]\!]_k = \bigvee_{i=0}^{k} p(\mathbf{s}_i)$$



- $\phi = \mathrm{G}\, p$

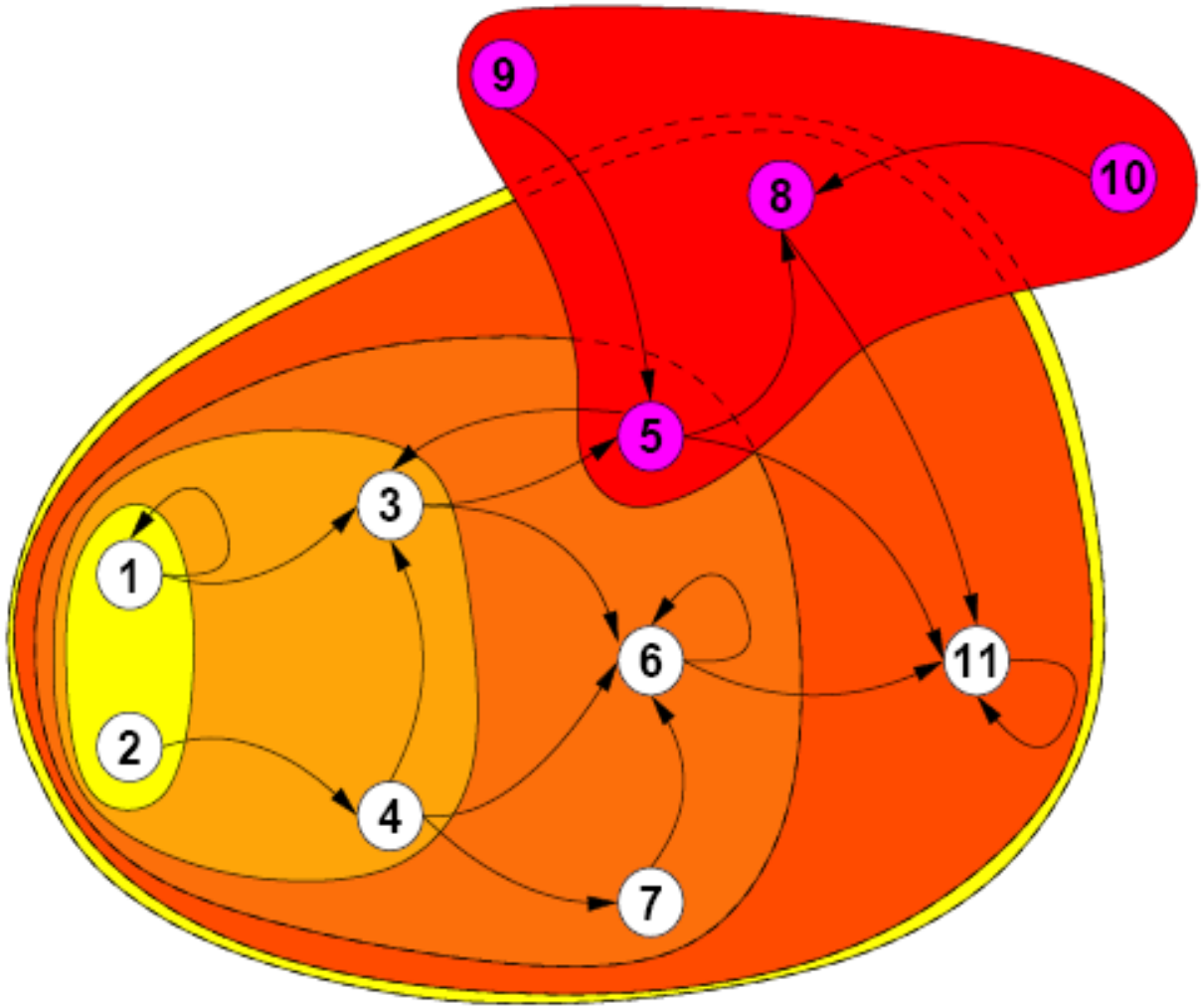
$$[\![\mathrm{G}\, p]\!]_k = \bigvee_{i=0}^{k} \left(\mathcal{T}(\mathbf{s}_k, \mathbf{s}_i) \wedge \bigwedge_{i=0}^{k} p(\mathbf{s}_i) \right)$$



SMC of Invariants

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

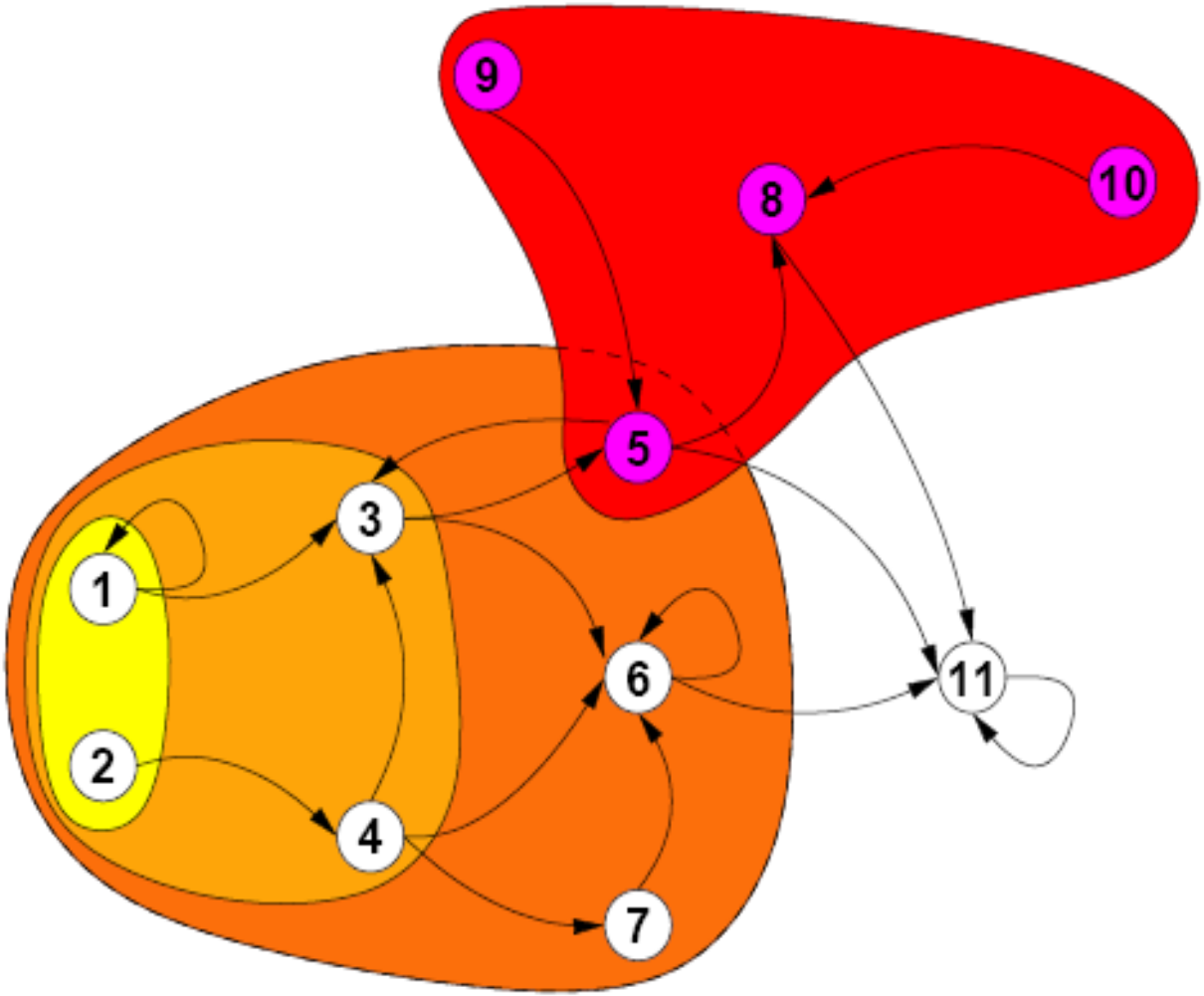
- is there a reachable state that is also a bad state (#)?



On the fly Checking of Invariants

Anticipate bug detection:

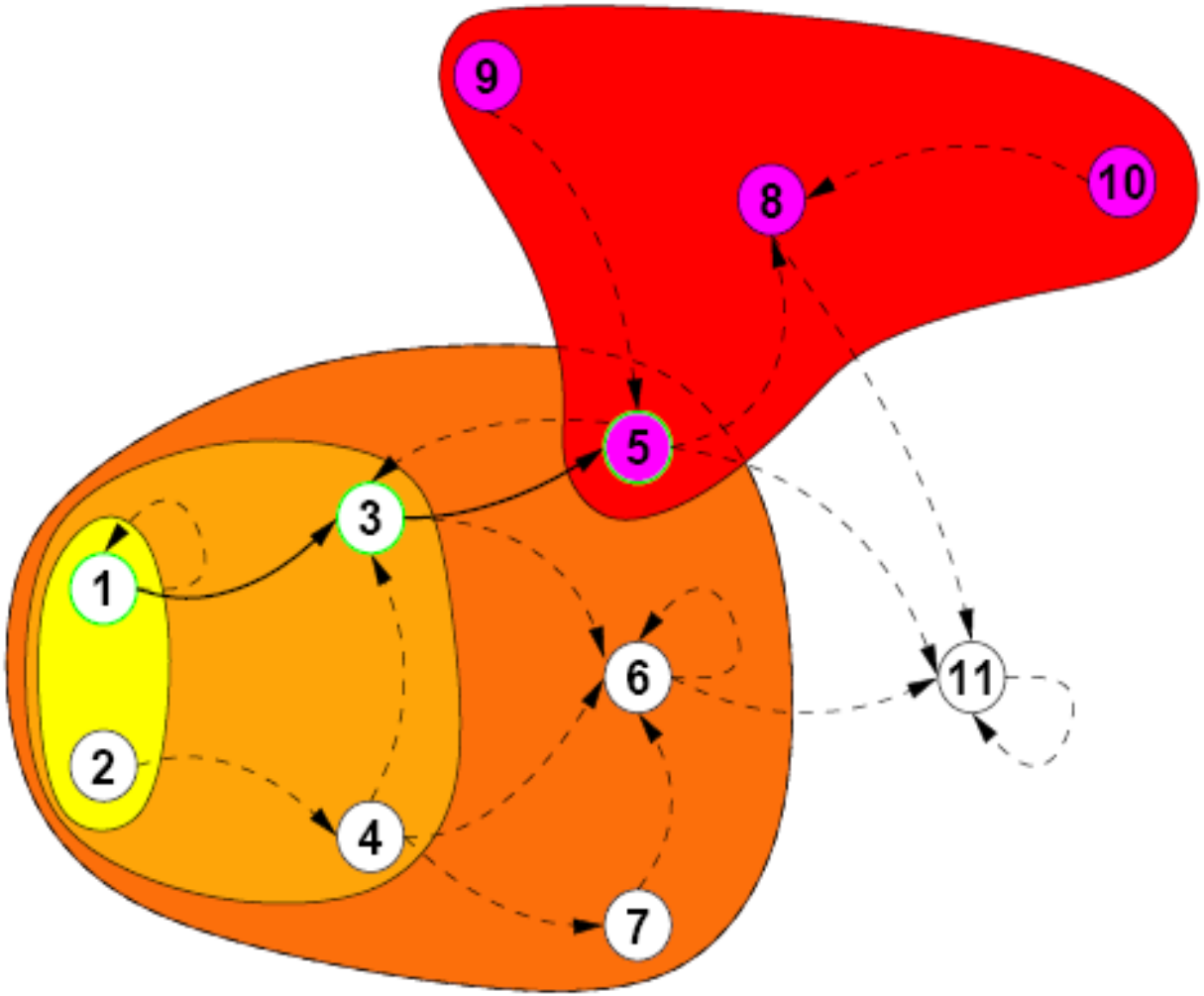
- at each layer, check if a new state is a bug



On the fly Checking of Invariants: Counterexamples

If a bug is found,

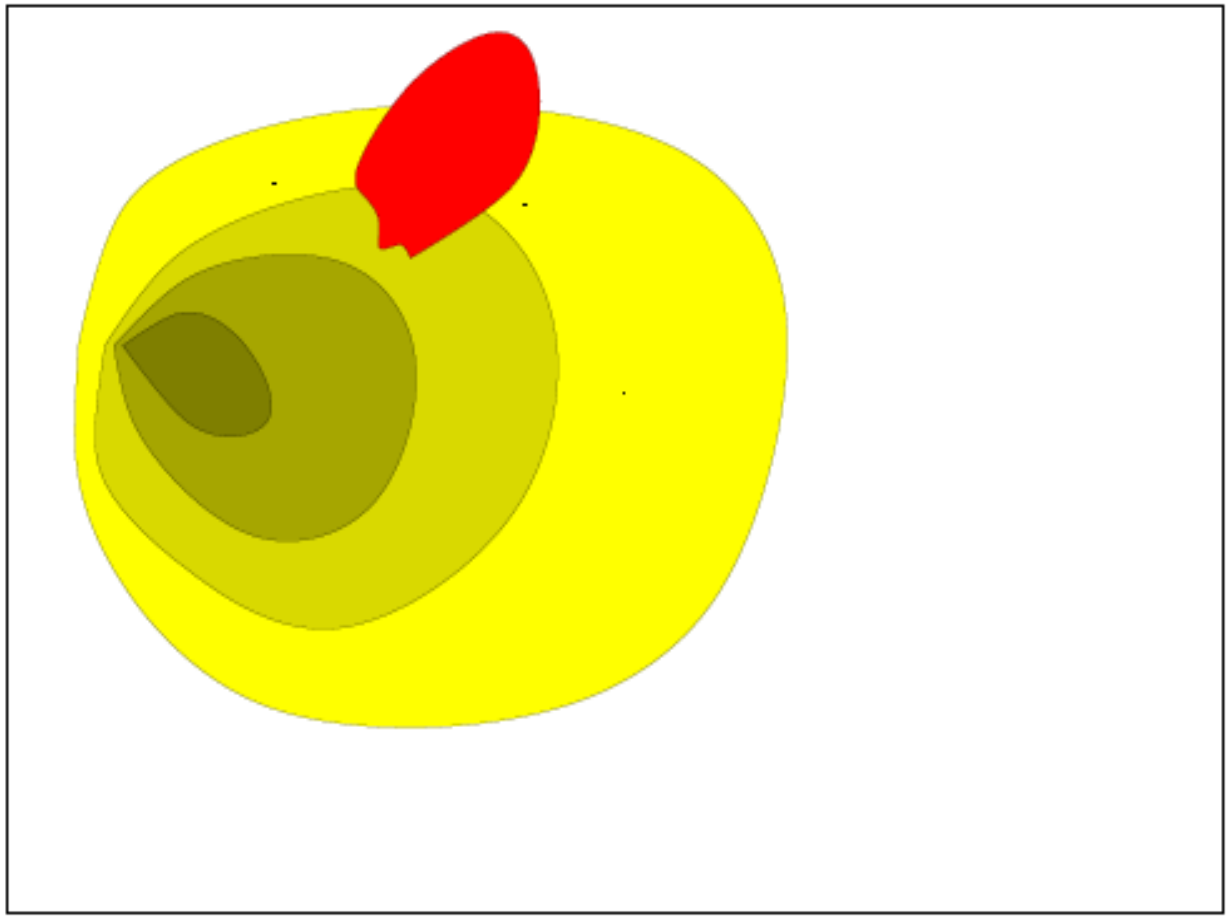
- a counterexample can be reconstructed proceeding backwards



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