TELECOM ParisTech

INSTITUT Mines-Télécom

Hardware Verification

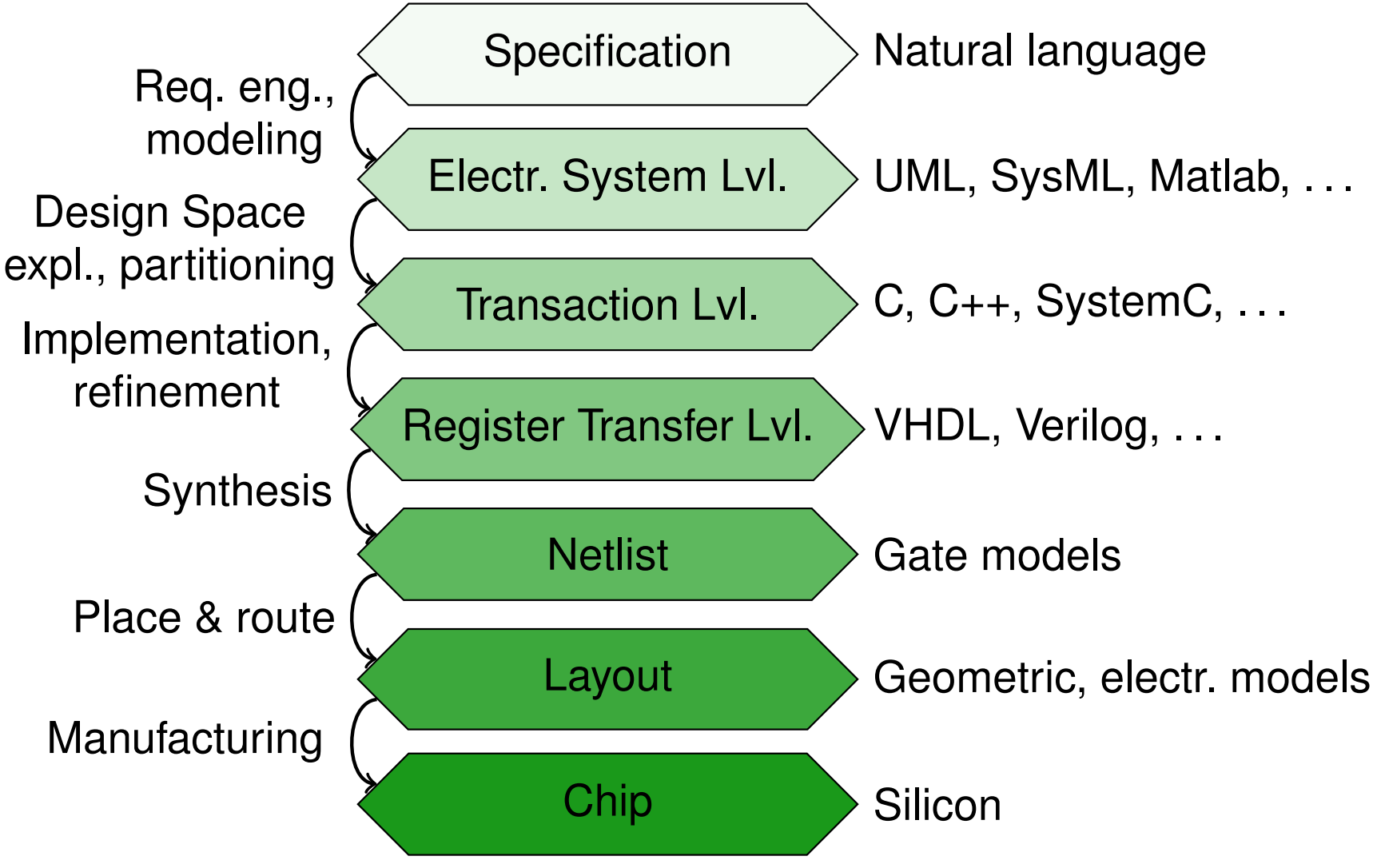
SE303b – Conception des systèmes sur puces (SoC)

Ulrich Kühne

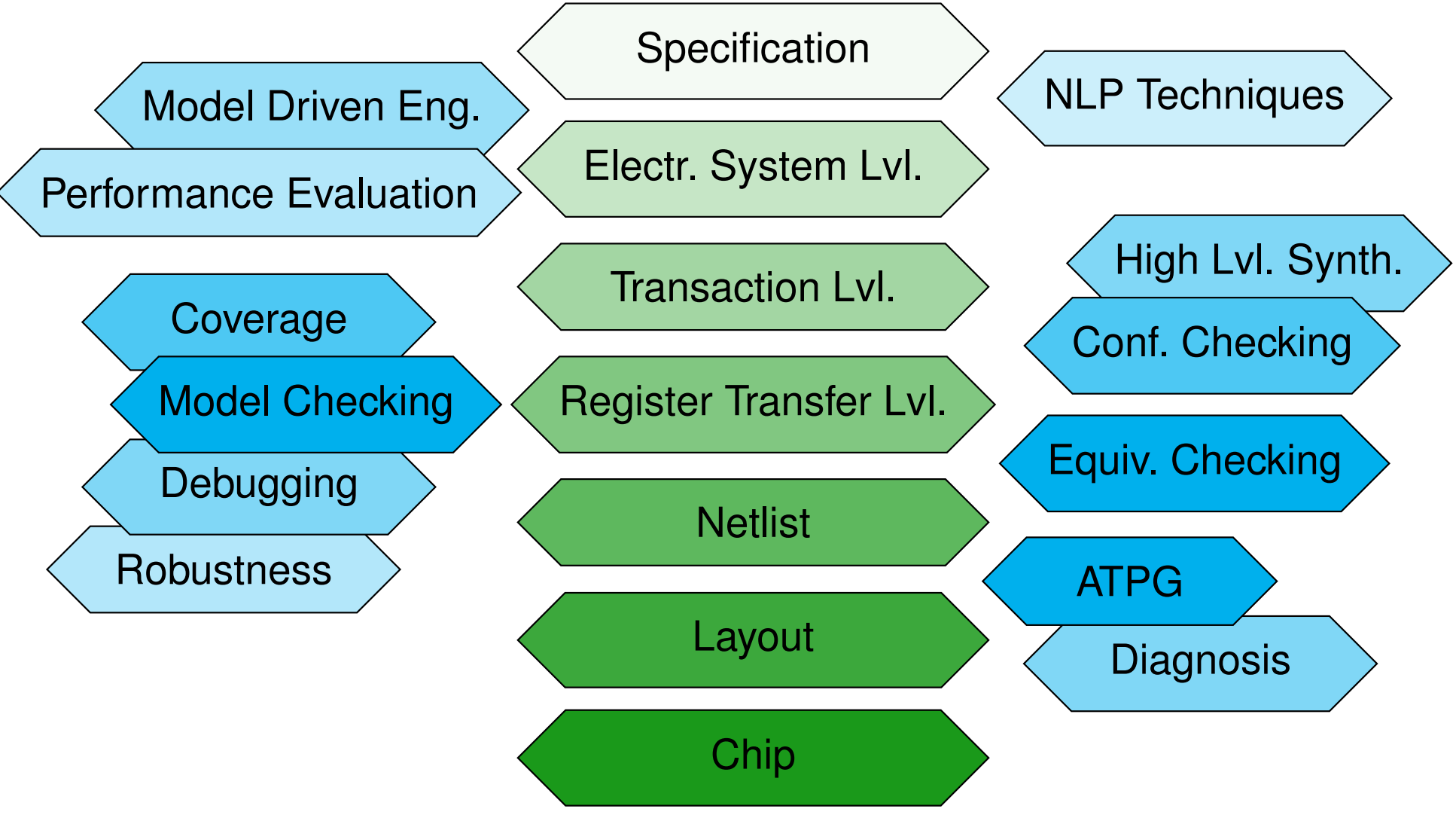
30/11/2018

Outline

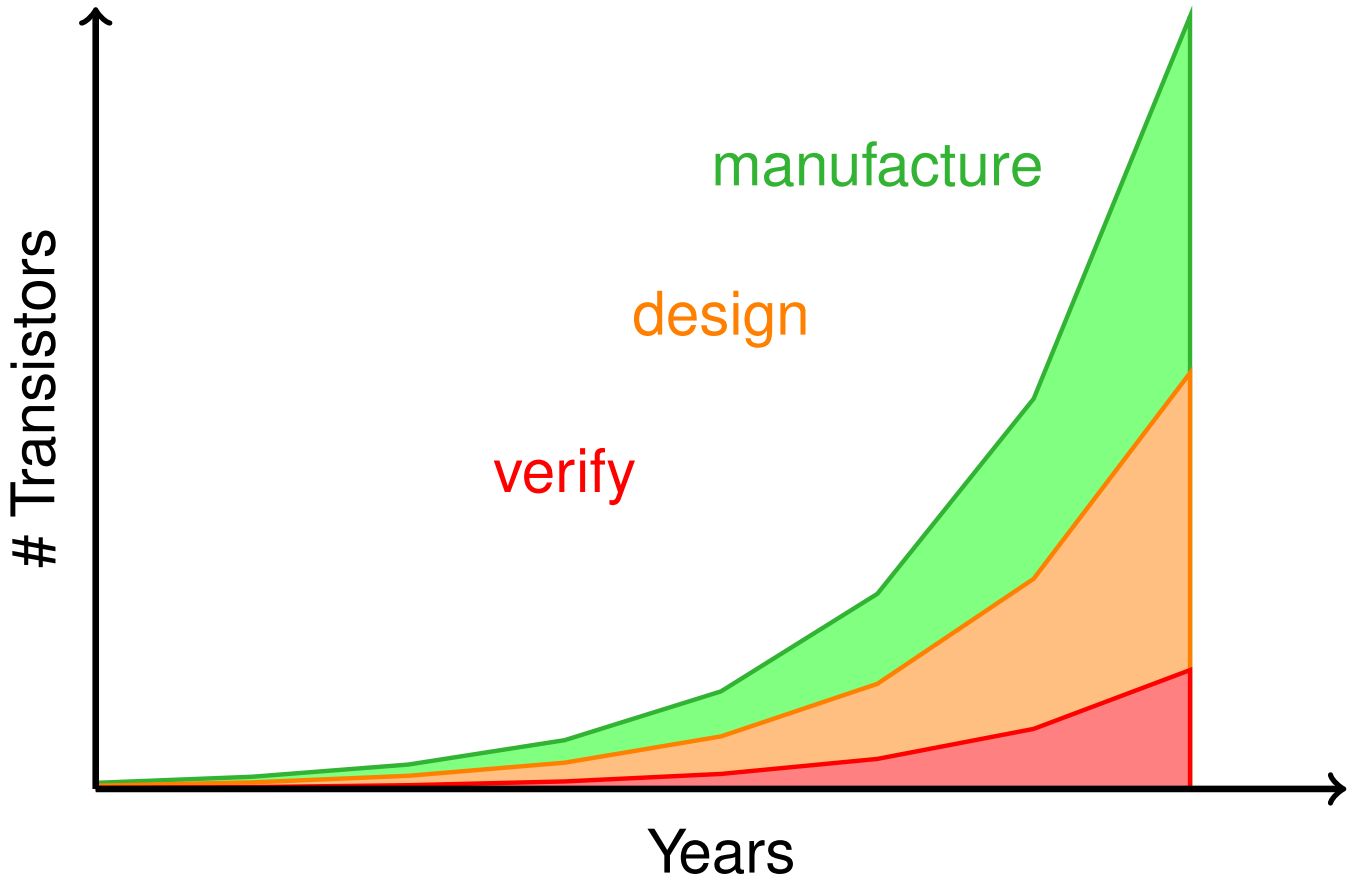
Hardware Design Flow



Hardware Verification Flow



Design Gap – Verification Gap

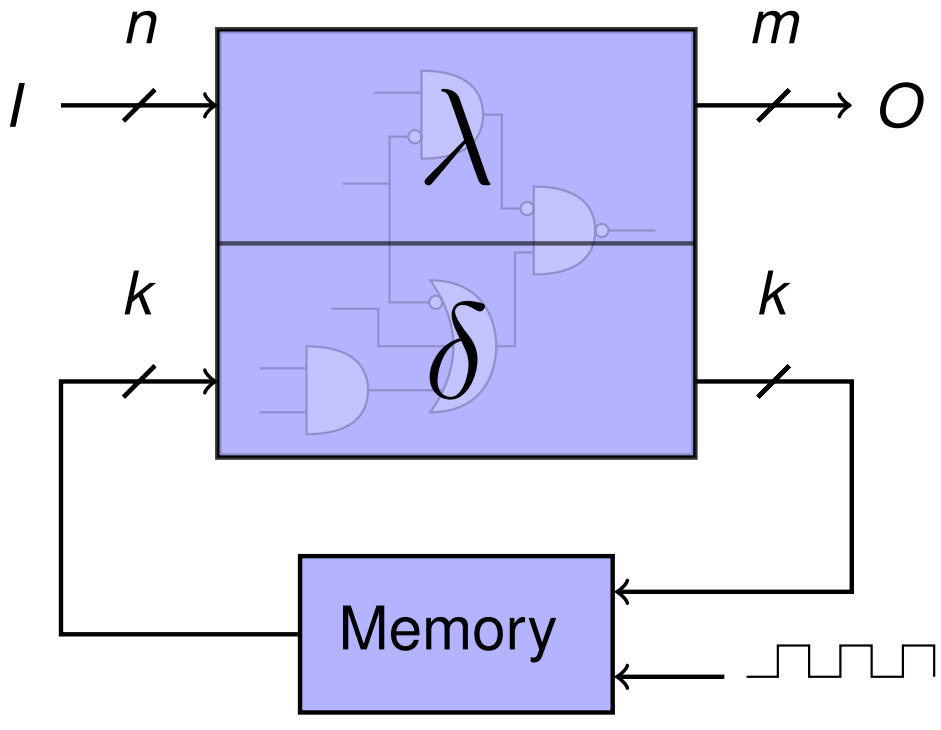


Outline

Functional Verification

- Dynamic verification (= simulation)
  still standard technology
- Pentium 4 overall simulated cycles < one minute
  at operation speed [Bentley, 2005]
- Full coverage is infeasible
- Increasing use of formal methods

Sequential Circuit Model



Mealy Machine:

$$\mathcal{M} = (I, O, S, S_0, \delta, \lambda)$$

$$\delta : S \times I \rightarrow S$$
$$\lambda : S \times I \rightarrow O$$
$$S_0 \subseteq S$$

$$I = \{0,1\}^n$$
$$O = \{0,1\}^m$$
$$S = \{0,1\}^k$$

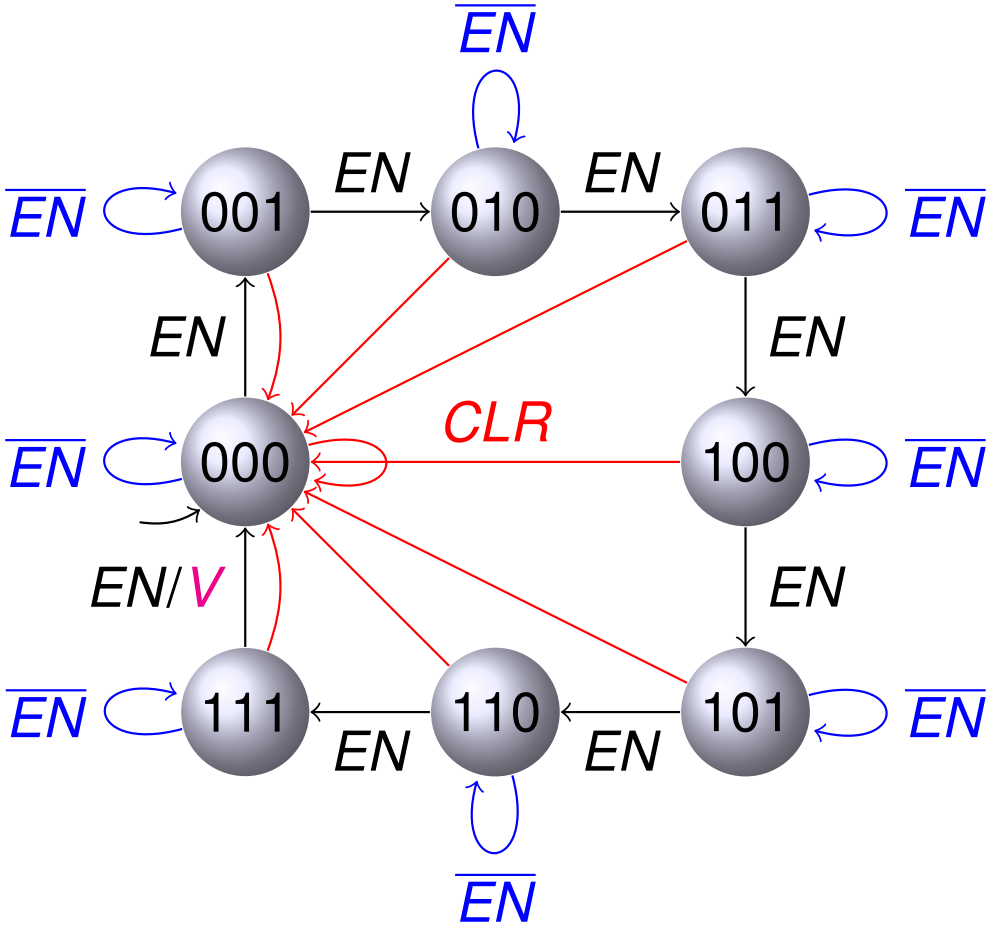
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input      CLK, EN, CLR;
  output reg S0, S1, S2;
  output     V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```

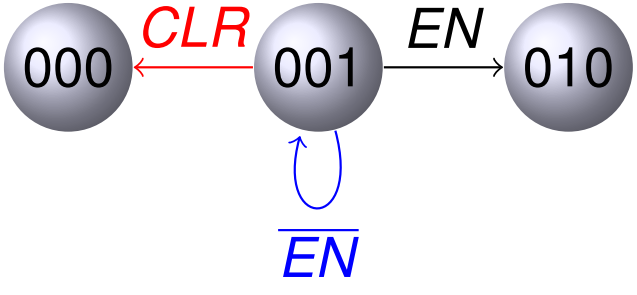


Verification Model

Mealy Machine:

$$\mathcal{M} = (I, O, S, S_0, \delta, \lambda)$$

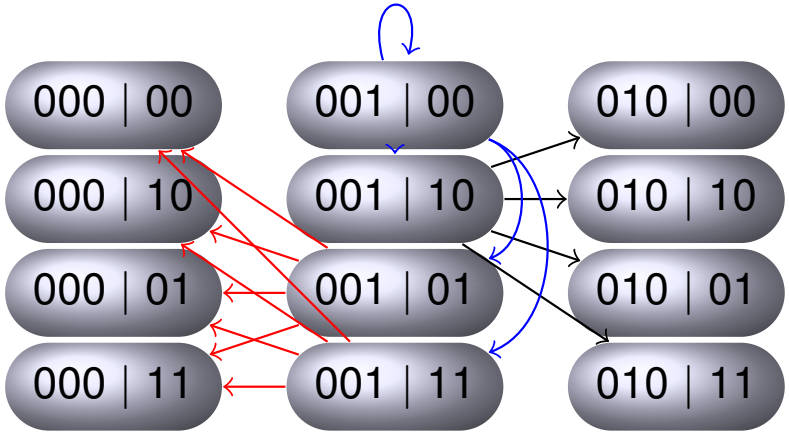
$$\delta : S \times I \to S$$
$$\lambda : S \times I \to O$$
$$S_0 \subseteq S$$

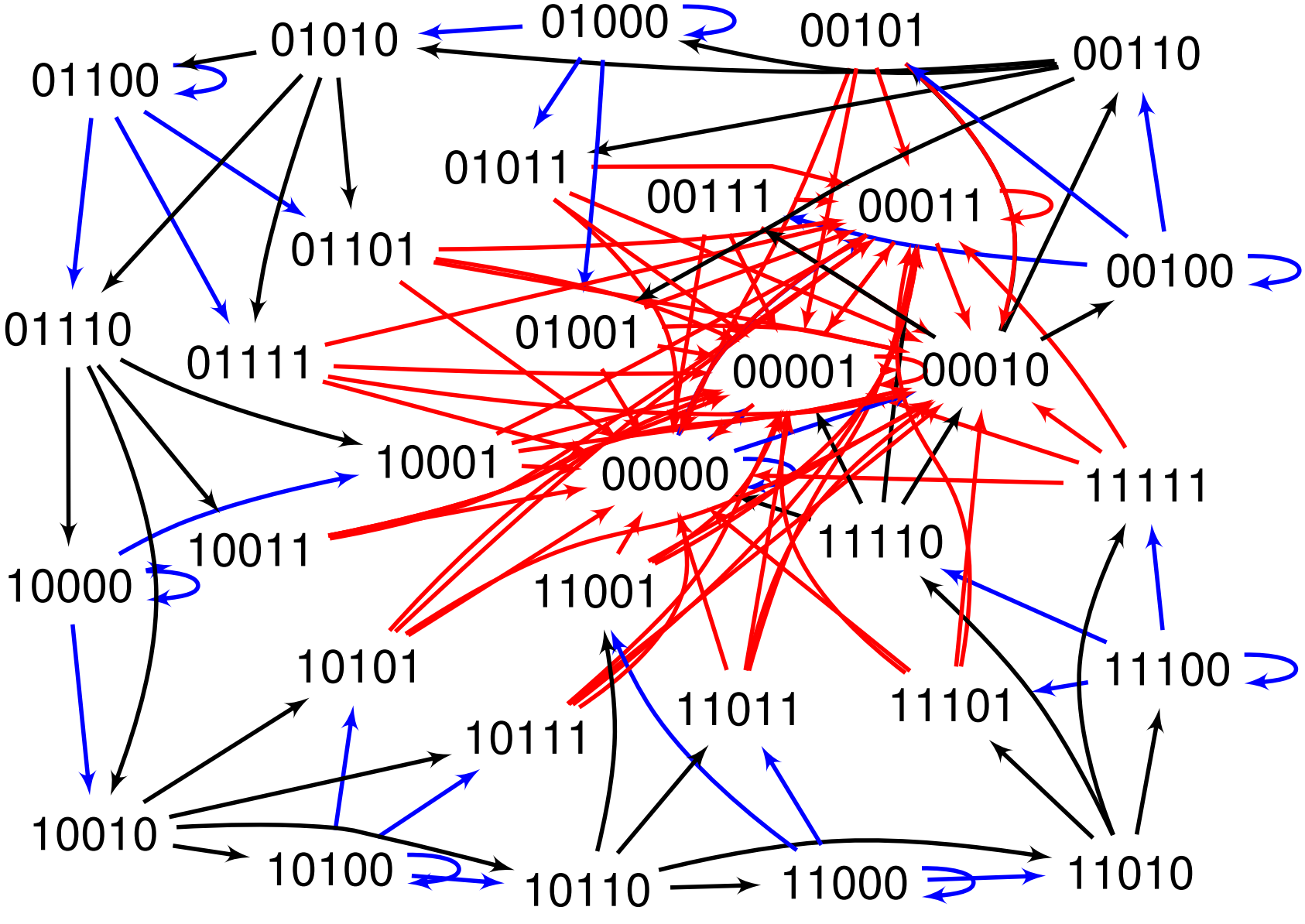


Kripke Structure:

$$\mathcal{K} = (S, S_0, \delta, \mathcal{V}, \mathcal{L})$$

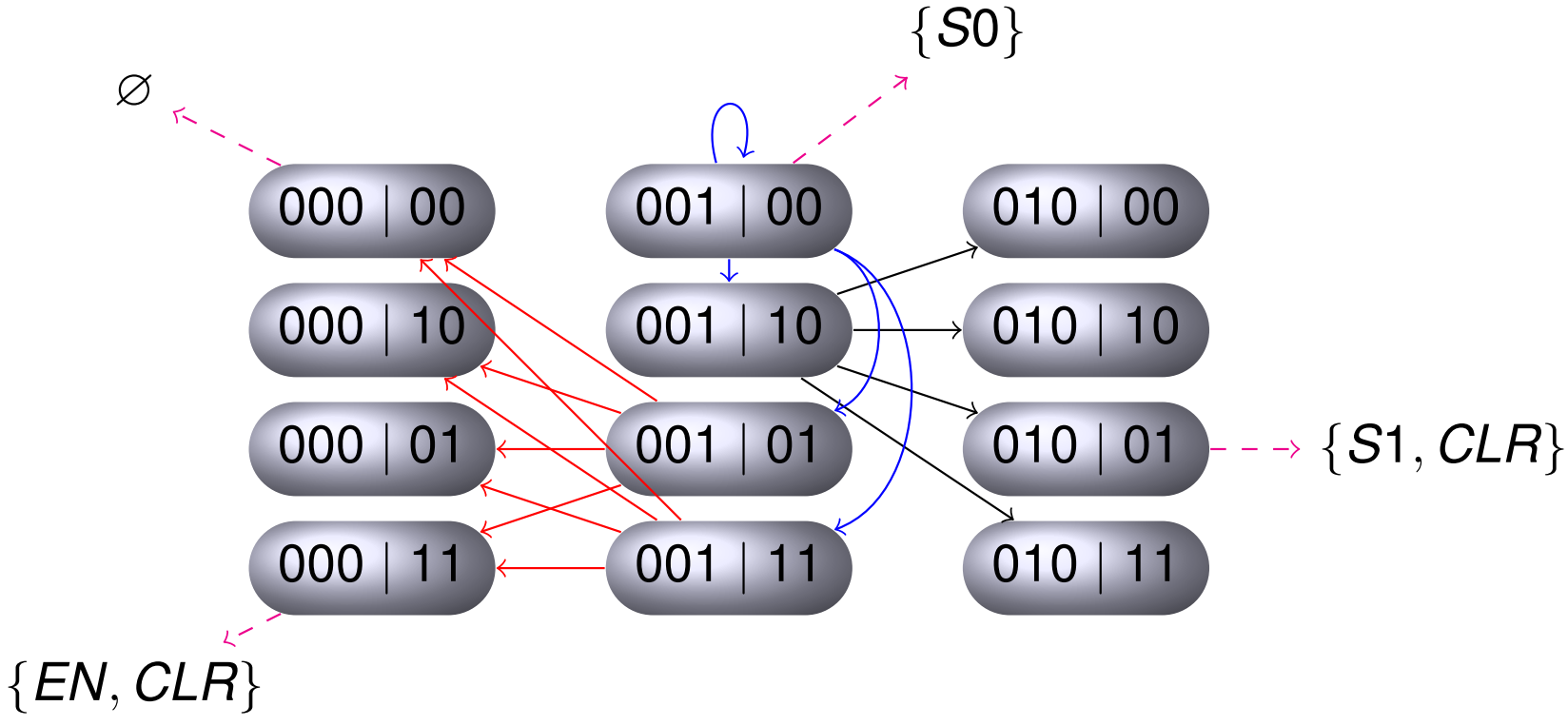
$\delta \subseteq S \times S$	transition relation
$S_0 \subseteq S$	initial states
$\mathcal{V}$	propositional variables
$\mathcal{L} : S \to 2^{\mathcal{V}}$	labelling function




01010
01000
00101
00110
01100
01011
00111
00011
01101
00100
01110
01001
01111
00001
00010
10001
00000
11111
10011
11110
10000
11001
10101
11100
11011
11101
10111
10010
10100
10110
11000
11010

Labelling Function

Propositional variables $\mathcal{V} = \{S2, S1, S0, EN, CLR, V\}$



What do we want to verify?

Safety

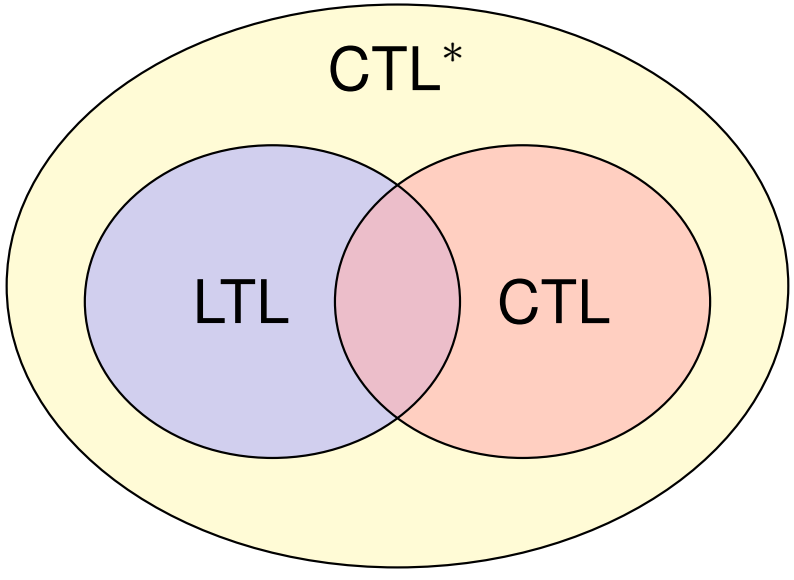
Something bad will never happen, e.g.
“The stack pointer will never overflow”
“The traffic lights will never be green at the same time”

Liveness

Something good will eventually happen, e.g.
“Every request will be granted”
“The cache and the main memory will eventually be consistent”

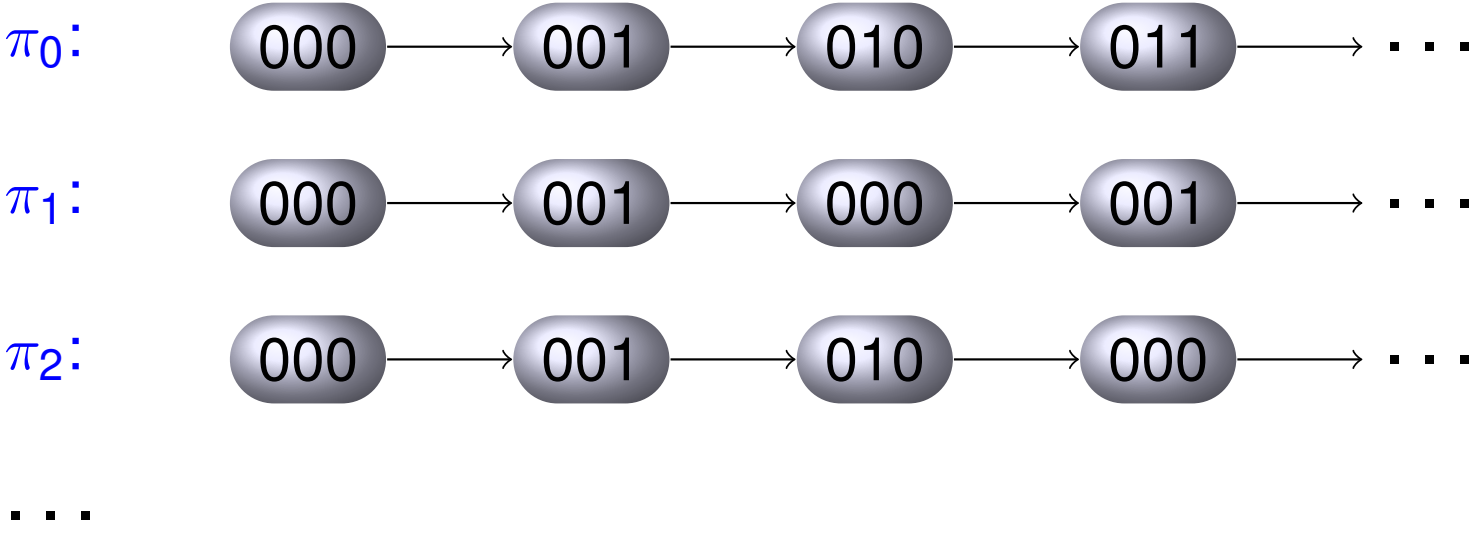
How to specify such properties?

- Temporal logic = propositional logic + time
- Discrete vs. continuous time
- Linear time view
- Branching time view

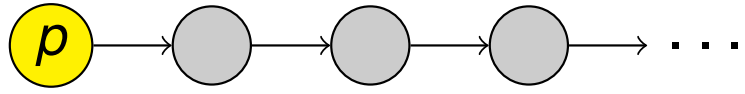
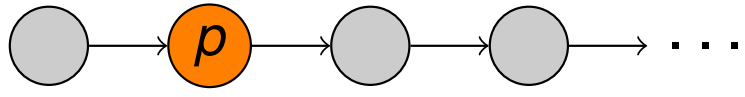
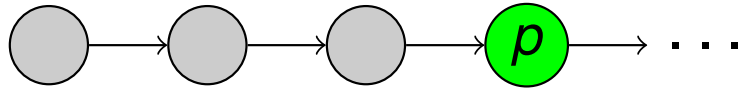
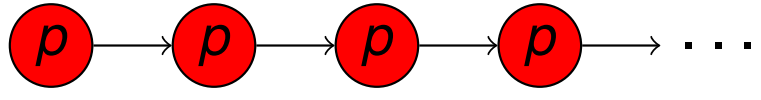
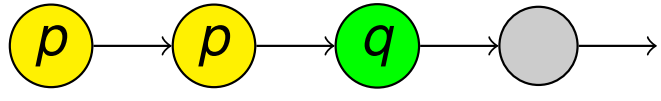


The Linear Time View

Computation paths



Linear Time Logic (LTL)

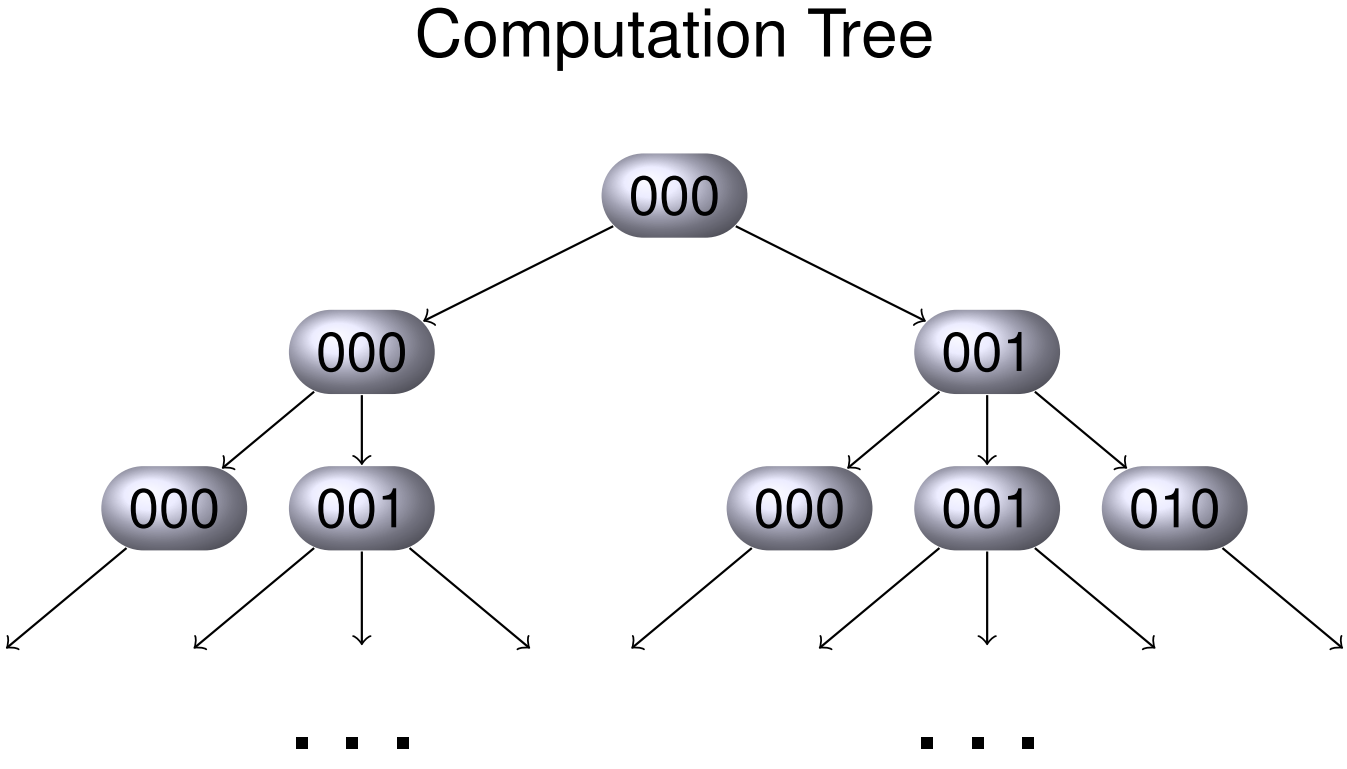
p holds (in the initial state)		p
p holds in the next state		$\mathbf{X}\, p$
p holds in the future		$\mathbf{F}\, p$
p holds globally		$\mathbf{G}\, p$
p holds until q		$p\ \mathbf{U}\ q$

Linear Time Logic (LTL)

An LTL formula over propositional variables $\mathcal{V}$ has the form

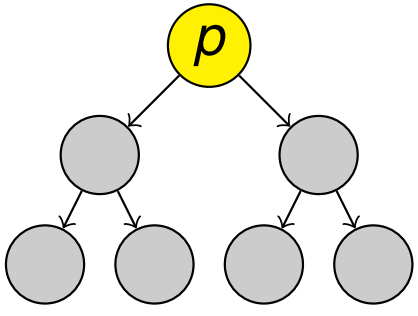
$$
\begin{array}{rll}
\mathit{LTL} ::= & p, & \text{where } p \in \mathcal{V} \\
\mid & \neg\varphi & \\
\mid & \varphi \wedge \psi & \\
\mid & \mathbf{X}\,\varphi & \\
\mid & \mathbf{F}\,\varphi & \\
\mid & \mathbf{G}\,\varphi & \\
\mid & \varphi\ \mathbf{U}\ \psi, & \text{where } \varphi, \psi \in \mathit{LTL}.
\end{array}
$$

Branching Time View



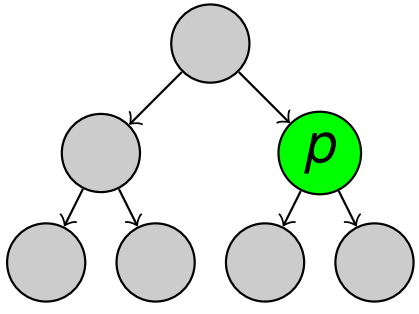
Computation Tree Logic (CTL)

Some property *p* holds
(in the initial state)



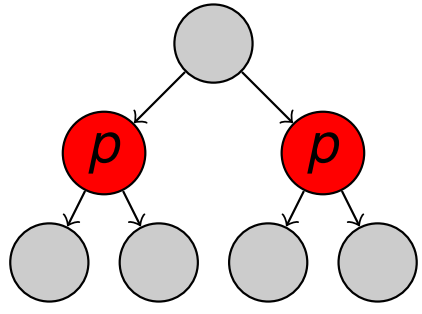
p

p holds in
some next state



EX *p*

p holds in
all next states



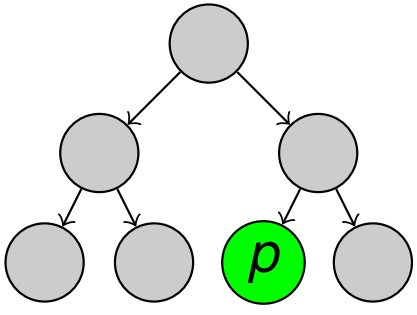
AX *p*

path
quantifier

next
operator

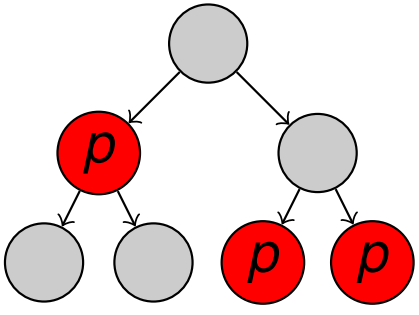
Further Modalities

p holds in some future state



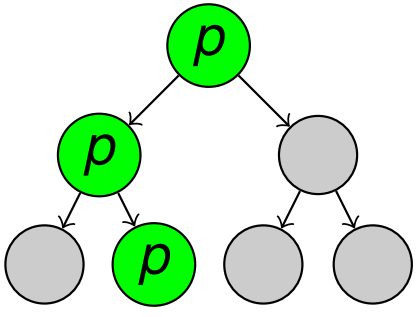
EF *p*

p holds eventually



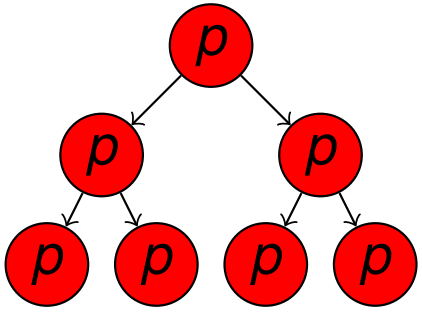
AF *p*

p holds globally on some path



EG *p*

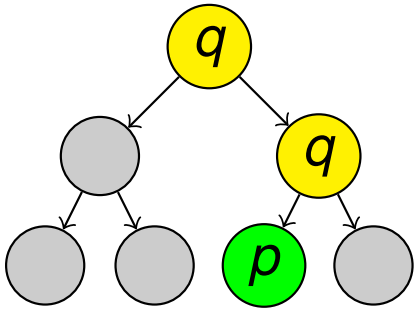
p holds globally on all paths



AG *p*

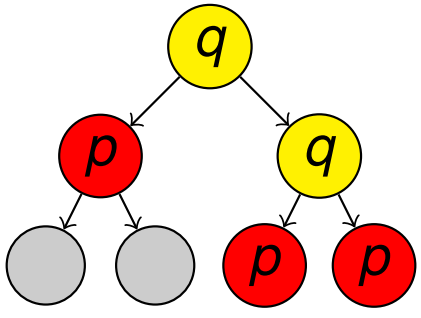
Until Modalities

On some path, q holds until p holds



E(p **U** q)

On all paths, q holds until p holds



A(p **U** q)

Computation Tree Logic (CTL)

A CTL formula over propositional variables $\mathcal{V}$ has the form

$$
\begin{array}{lll}
\mathit{CTL} ::= & p, \quad \text{where } p \in \mathcal{V} & \\
\mid & \varphi \wedge \psi & \mid \quad \neg\varphi \\
\mid & \mathbf{EX}\ \varphi & \mid \quad \mathbf{AX}\ \varphi \\
\mid & \mathbf{EF}\ \varphi & \mid \quad \mathbf{AF}\ \varphi \\
\mid & \mathbf{EG}\ \varphi & \mid \quad \mathbf{AG}\ \varphi \\
\mid & \mathbf{E}(\varphi\ \mathbf{U}\ \psi) & \mid \quad \mathbf{A}(\varphi\ \mathbf{U}\ \psi), \quad \text{where } \varphi, \psi \in \mathit{CTL}
\end{array}
$$

What do we want to verify?

Safety

"The stack pointer will never overflow" $\mathbf{AG}\ (\mathrm{sp} < 4096)$

"The traffic lights will never be green at the same time" $\neg\mathbf{EF}\ (\mathrm{tl}_1 \wedge \mathrm{tl}_2)$

Liveness

"Every request will be granted" $\mathbf{AG}\ (\mathrm{req} \rightarrow \mathbf{AF}\ \mathrm{gnt})$

"The cache and the main memory will eventually be consistent" $\mathbf{AF}\ (\mathrm{mem}_i = \mathrm{cache}_i)$

Model Checking

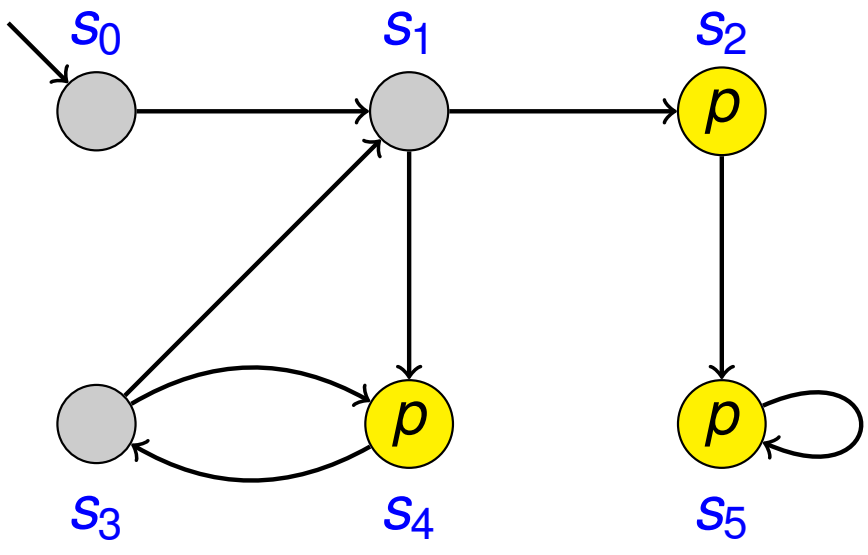
Model Checking

Given a Kripke Structure $\mathcal{K}$ and a CTL formula φ,
check if $\mathcal{K} \models \varphi$.

How do we do this?

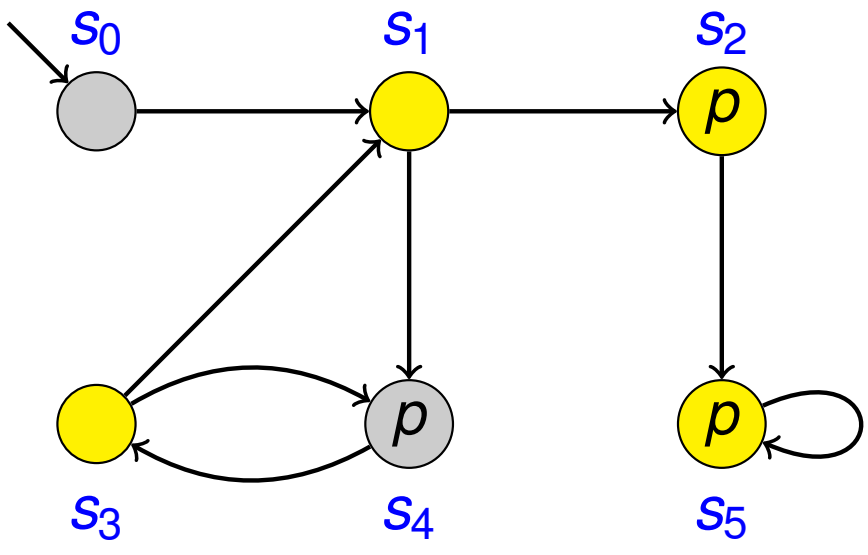
1. Compute all states in which φ holds:
   $\tau(\varphi) = \{s \in S \mid \mathcal{K}, s \models \varphi\}$
2. Check if the initial states are a subset of those states:
   $S_0 \setminus \tau(\varphi) = \varnothing$

Example



$$\tau(p) = \{s_2, s_4, s_5\}$$

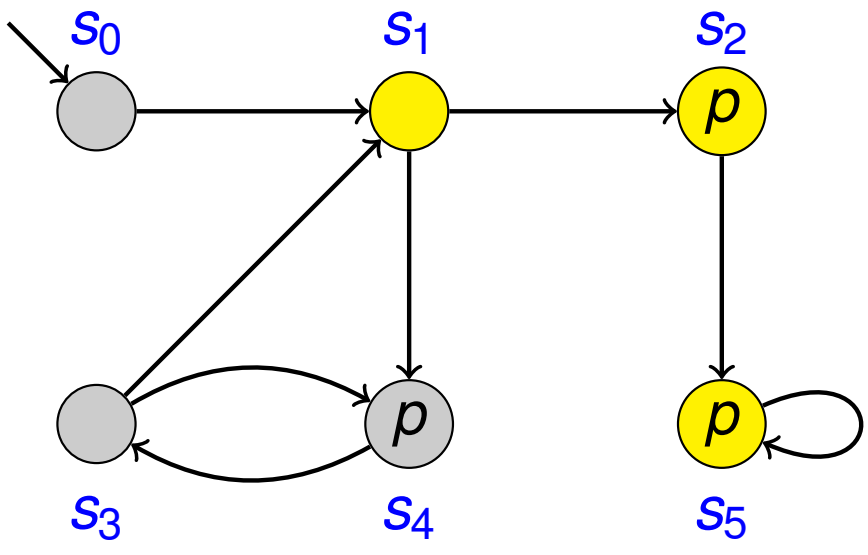
Example



$$\tau(p) = \{s_2, s_4, s_5\}$$

$$\tau(\mathbf{EX}\ p) = \{s_1, s_2, s_3, s_5\}$$

Example

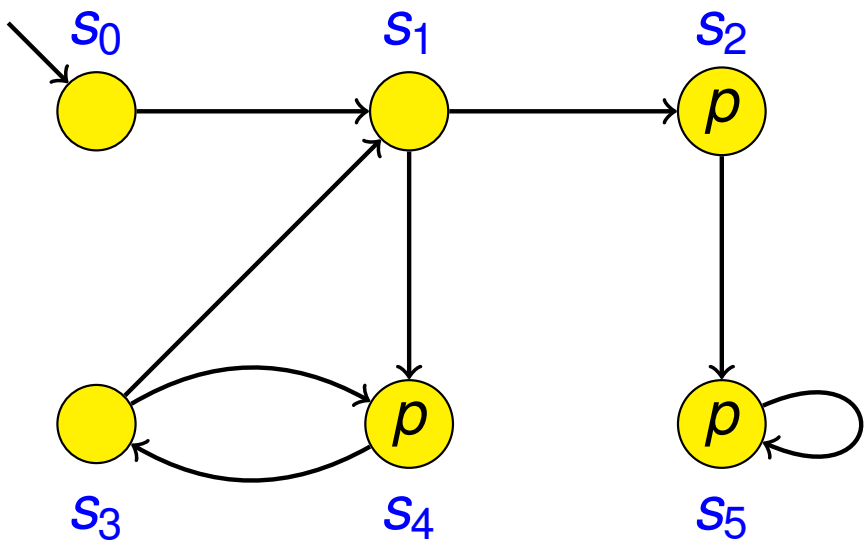


$$\tau(p) = \{s_2, s_4, s_5\}$$

$$\tau(\mathbf{EX}\ p) = \{s_1, s_2, s_3, s_5\}$$

$$\tau(\mathbf{AX}\ p) = \{s_1, s_2, s_5\}$$

Example



$$\tau(p) = \{s_2, s_4, s_5\}$$

$$\tau(\mathbf{EX}\ p) = \{s_1, s_2, s_3, s_5\}$$

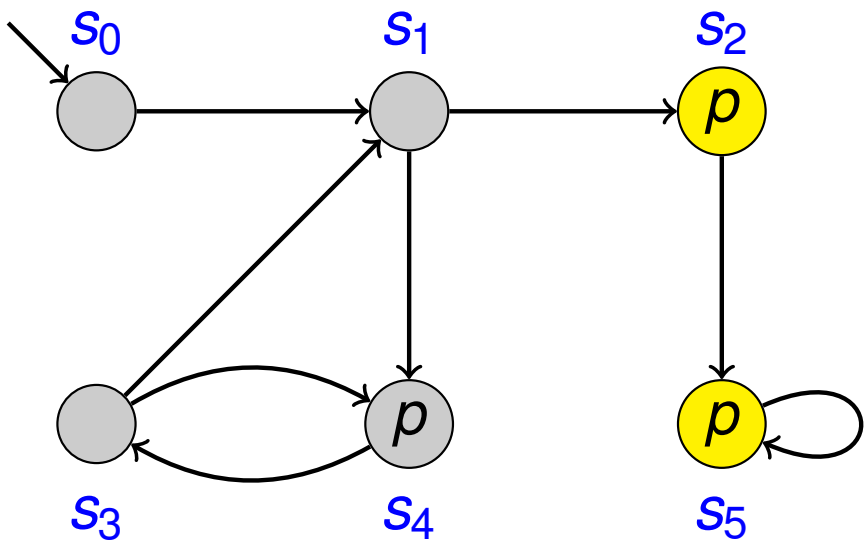
$$\tau(\mathbf{AX}\ p) = \{s_1, s_2, s_5\}$$

$$\tau(\mathbf{EF}\ p) = \{s_2, s_4, s_5\} \cup \{s_1, s_3\} \cup \{s_0\}$$

Expansion rules:

$$\mathbf{EF}\ \varphi = \varphi \vee \mathbf{EX}\ \mathbf{EF}\ \varphi$$

Example



$$\tau(p) = \{s_2, s_4, s_5\}$$

$$\tau(\mathbf{EX}\ p) = \{s_1, s_2, s_3, s_5\}$$

$$\tau(\mathbf{AX}\ p) = \{s_1, s_2, s_5\}$$

$$\tau(\mathbf{EF}\ p) = \{s_2, s_4, s_5\} \cup \{s_1, s_3\} \cup \{s_0\}$$

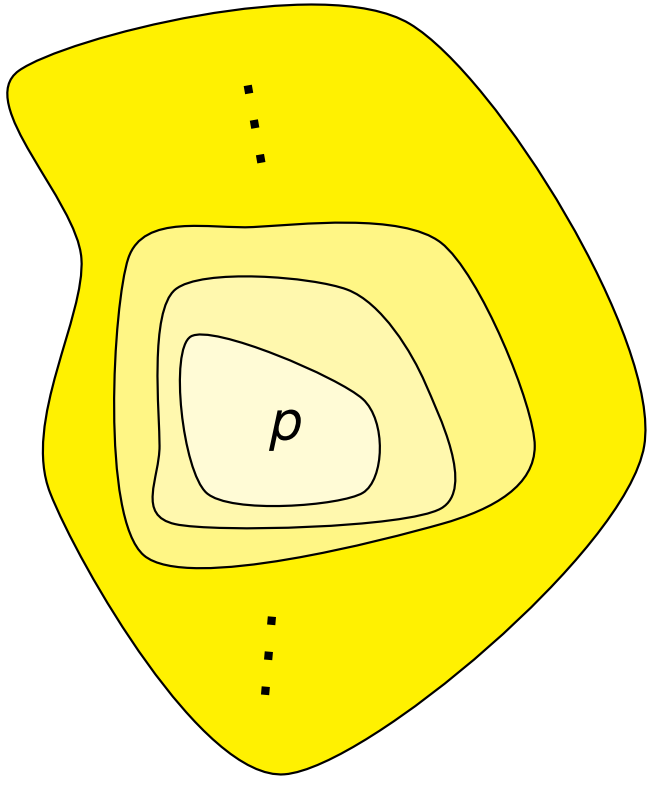
$$\tau(\mathbf{AG}\ p) = \{s_2, s_4, s_5\} \cap \{s_2, s_5\}$$

Expansion rules:

$$\mathbf{EF}\ \varphi = \varphi \vee \mathbf{EX}\ \mathbf{EF}\ \varphi$$

$$\mathbf{AG}\ \varphi = \varphi \wedge \mathbf{AX}\ \mathbf{AG}\ \varphi$$

Fixed Point Algorithm for EF p



$$S_0 = p$$
$$S_1 = p \cup \mathbf{EX}\ p$$
$$S_2 = p \cup \mathbf{EX}\ p \cup \mathbf{EX}\ \mathbf{EX}\ p$$
$$\ldots$$
$$S_n = p \cup \bigcup_{i=1}^{n} \mathbf{EX}^i p = S_{n-1}$$

$$\Rightarrow S_n = \tau(\mathbf{EF}\ p)$$

Fixed Points

Let $f : \mathcal{P}(S) \to \mathcal{P}(S)$ a set-valued function and $Z \subseteq G$.

- Z is called a fixed point of f if $f(Z) = Z$
- Z is the least fixed point of f it is a fixed point and for all other fixed points U of f it holds that $Z \subseteq U$.
- Z is the greatest fixed point of f it is a fixed point and for all other fixed points U of f it holds that $U \subseteq Z$.

Fixed Points (2)

A function $f : \mathcal{P}(S) \to \mathcal{P}(S)$ is called monotone if for all $X, Y \subseteq S$

$$X \subseteq Y \Rightarrow f(X) \subseteq f(Y) \tag{1}$$

Knaster-Tarski Theorem

Let $f : \mathcal{P}(S) \to \mathcal{P}(S)$ be a monotone function. Then f has a least and a greatest fixed point.

- $\bigcup_{n \geq 1} f^n(\varnothing)$ is the least fixed point of f.
- $\bigcap_{n \geq 1} f^n(S)$ is the greatest fixed point of f.

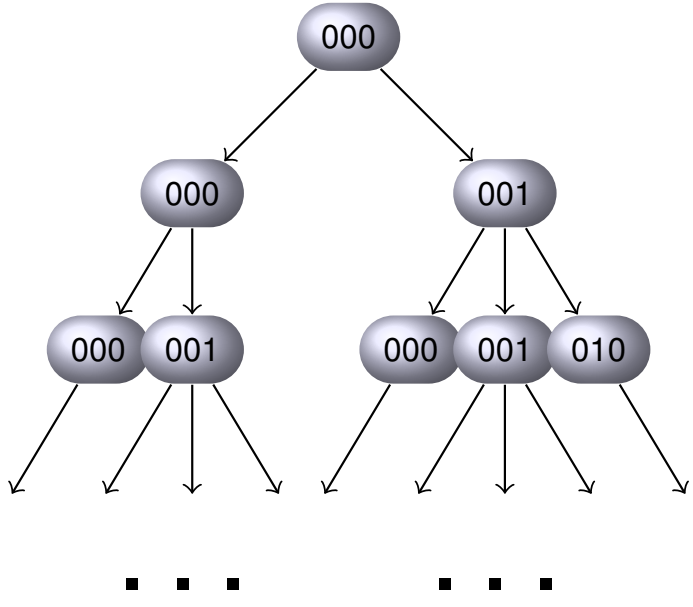
CTL Model Checking

Let $\mathcal{K} = (S, S_0, \delta, \mathcal{V}, \mathcal{L})$ be a Kripke structure.

$$
\begin{array}{lcl}
\tau(p) & = & \{s \in S \mid p \in \mathcal{L}(s)\} \\
\tau(\varphi \wedge \psi) & = & \tau(\varphi) \cap \tau(\psi) \\
\tau(\varphi \vee \psi) & = & \tau(\varphi) \cup \tau(\psi) \\
\tau(\neg\varphi) & = & S \setminus \tau(\varphi) \\
\tau(\mathbf{EF}\ \varphi) & = & \mathrm{lfp} Z.\ \tau(\varphi) \cup \mathbf{EX}\ (Z) \\
\tau(\mathbf{AF}\ \varphi) & = & \mathrm{lfp} Z.\ \tau(\varphi) \cup \mathbf{AX}\ (Z) \\
\tau(\mathbf{EG}\ \varphi) & = & \mathrm{gfp} Z.\ \tau(\varphi) \cap \mathbf{EX}\ (Z) \\
\tau(\mathbf{AG}\ \varphi) & = & \mathrm{gfp} Z.\ \tau(\varphi) \cap \mathbf{AX}\ (Z) \\
\tau(\mathbf{E}(\varphi\ \mathbf{U}\ \psi)) & = & \mathrm{lfp} Z.\ \tau(\psi) \cup (\tau(\varphi) \cap \mathbf{EX}\ (Z)) \\
\tau(\mathbf{A}(\varphi\ \mathbf{U}\ \psi)) & = & \mathrm{lfp} Z.\ \tau(\psi) \cup (\tau(\varphi) \cap \mathbf{AX}\ (Z))
\end{array}
$$

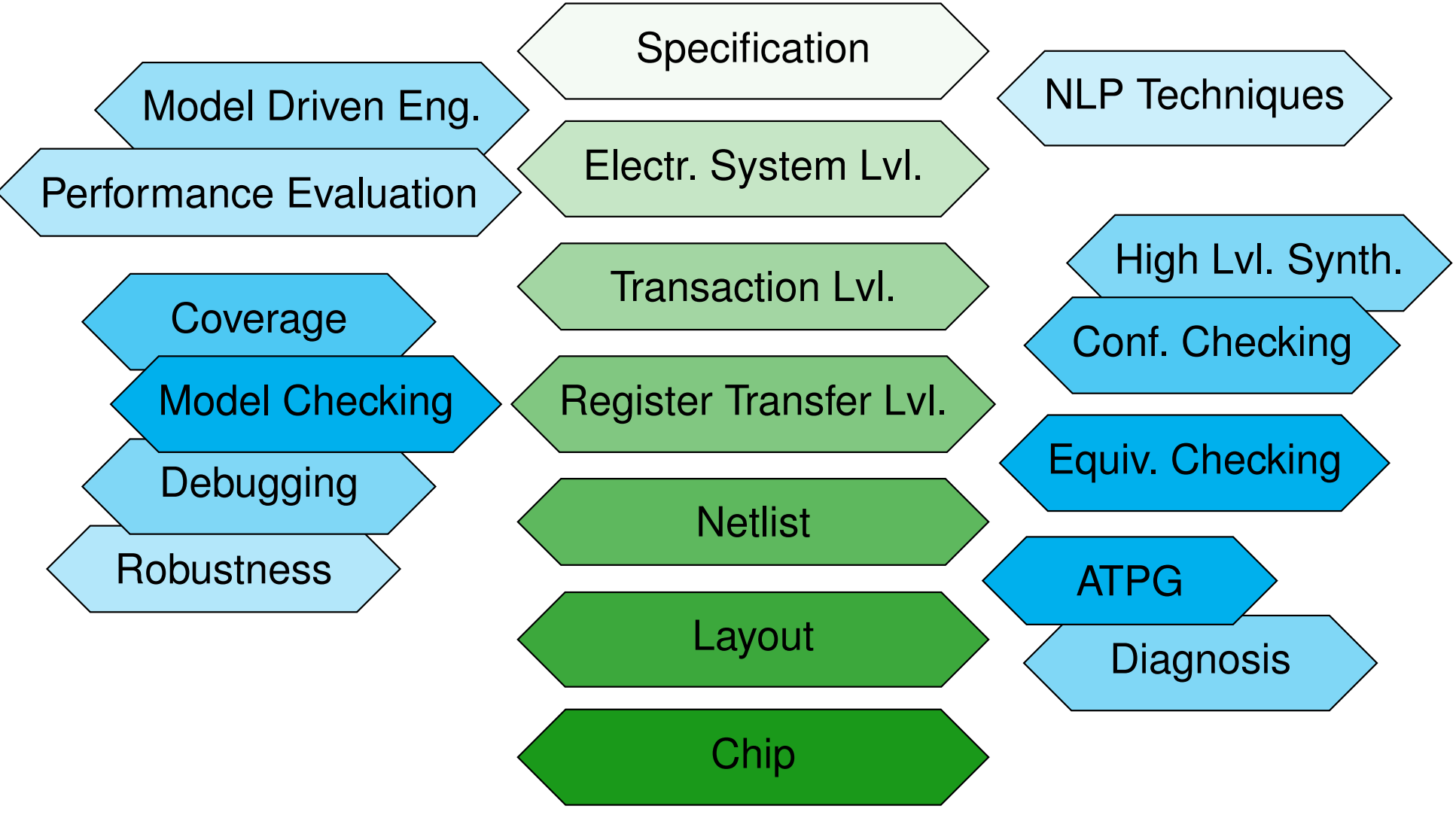
Model Checking

- Complexity depends heavily on state space
- Need for efficient data structures
- State space explosion still a problem
- Works for small to medium (or very regular) systems
- Popular tool: NuSMV [Cimatti et al., 2002]
- Ongoing research



Outline

Hardware Design Flow



Hardware Verification vs Test

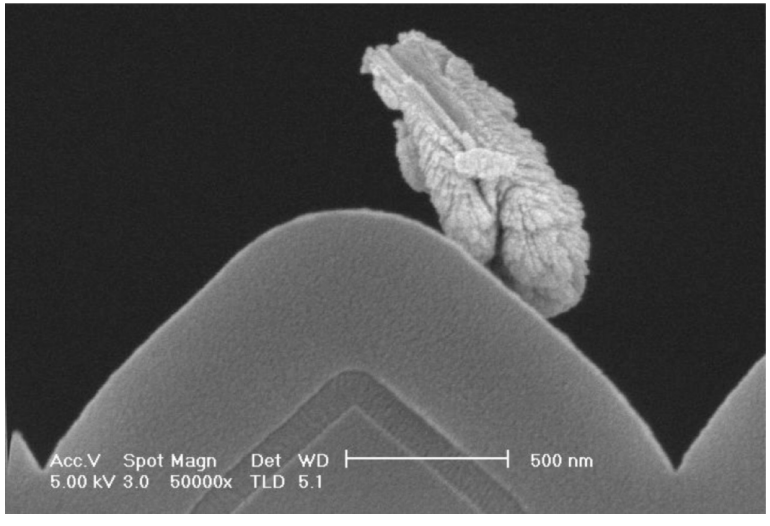
Verification

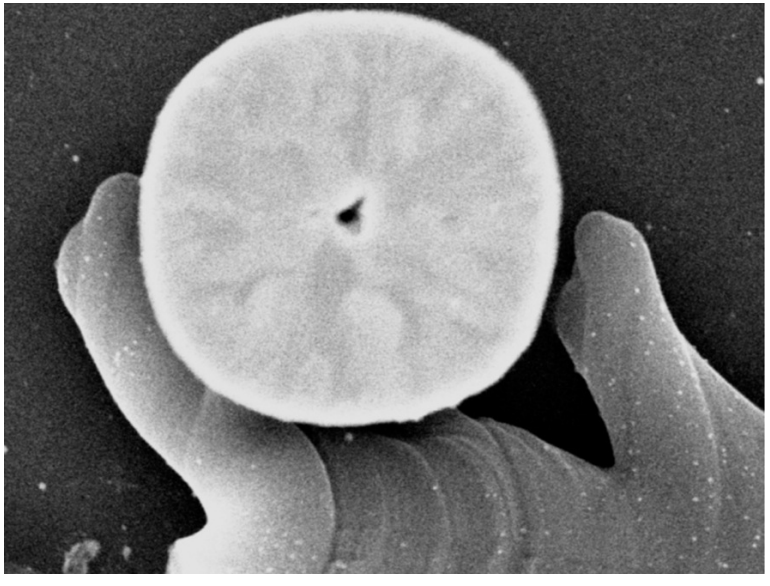
- Detect design bugs
- Extract properties from requirements
- Applied on RTL code
- High manual effort

Test

- Detect physical defects
- Test generation from netlist according to fault model
- Applied on fabricated chips
- High automation

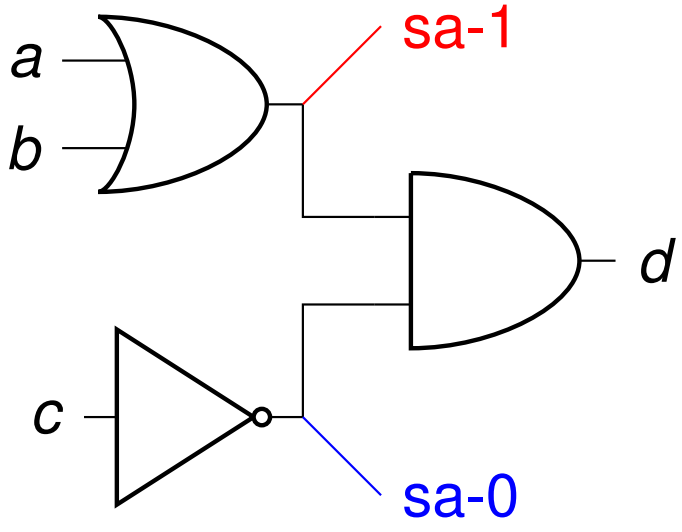
Physical Defects





[Source: IEEE Spectrum “The Art of Failure”]

Stuck-at Fault Model



a	b	c	d
0	0	0	0 0
0	0	**0/1**	
0	0	1	0
0	1	0	1**1/0**
0	1	1	0
1	0	0	1**1/0**
1	0	1	0
1	1	0	1**1/0**
1	1	1	0

- $\langle 000 \rangle$ is a test vector for the shown stuck-at-1 fault
- $\{\langle 010 \rangle, \langle 100 \rangle, \langle 110 \rangle\}$ are test vectors for the stuck-at-0 fault

Automatic Test Pattern Generation

ATPG

- Create a list of all possible (stuck-at) faults
- For each fault:
  - Find a test pattern
  - Drop all other faults detected by this pattern

- Untestable faults?
- Hard to test faults?
- Sequential tests?
- Test compression?

References I

Bentley, B. (2005).
Validating a modern microprocessor.
In Etessami, K. and Rajamani, S., editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 2–4. Springer Berlin Heidelberg.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002).
NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking.
In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark. Springer.