Formal Verification

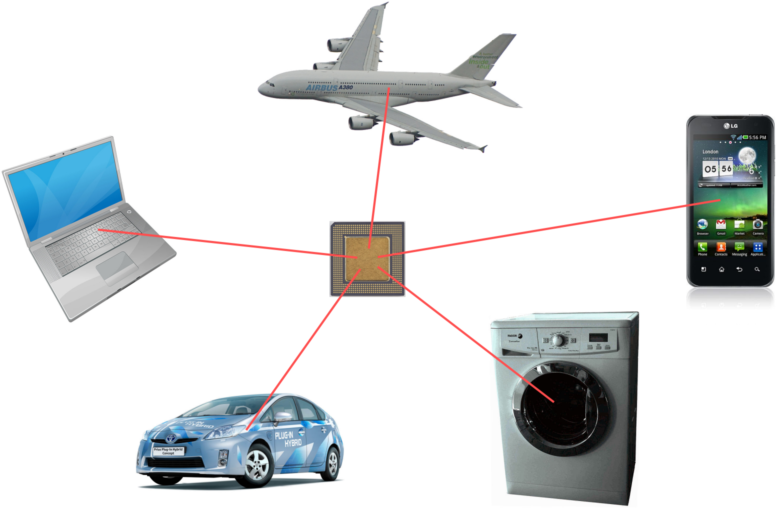
SE303 – Conception des systèmes sur puces (SoC)

Ulrich Kühne

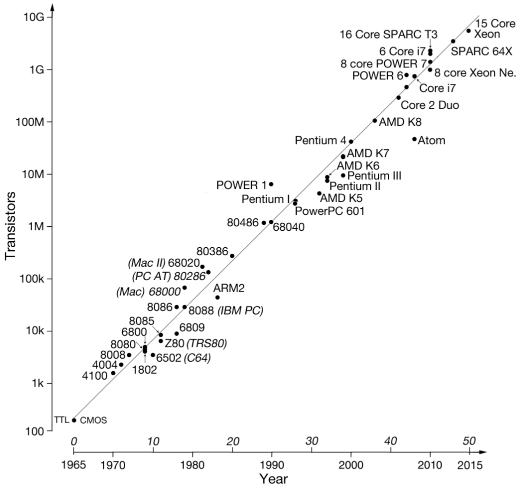
29/11/2017

Outline

Motivation



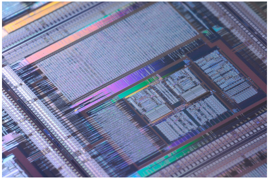
Motivation



[Source: www.elektormagazine.com/articles/moores-law]

TELECOM ParisTech

Motivation

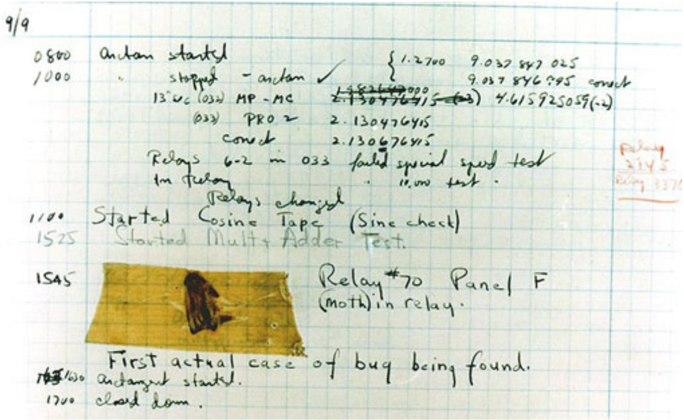


[cc photo by mark.sze]

- Transistor count of > 3 billion
- Gate level models are huge
- Big designer teams (several hundreds)
- Big correctness issues
- Late bugs are extremely expensive

Motivation

The First Bug (1947)



[Photo: U.S. Naval Historical Center]

Motivation

The Pentium FDIV Bug (1994)

let $x = 4195835, \; y = 3145727 \quad \Rightarrow \quad x - \frac{x}{y} \cdot y =$ 256

- Bug in floating point unit $\Rightarrow$ \$ 450 Mio. loss for Intel

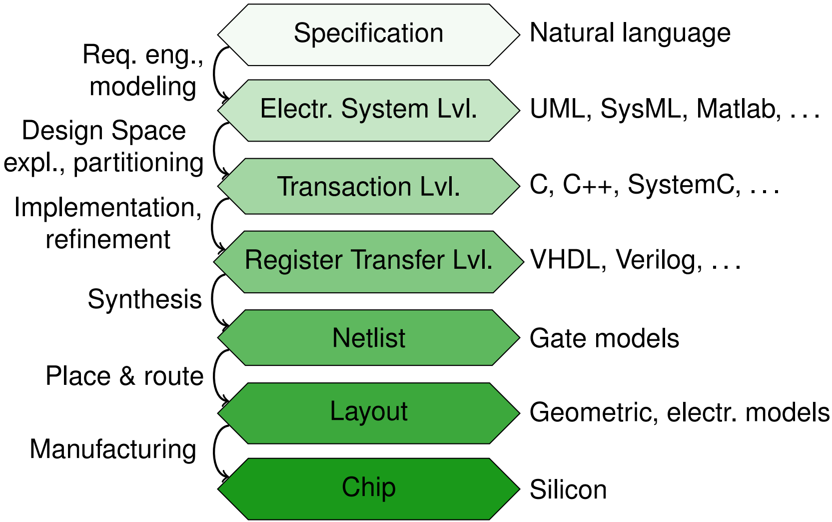
820 Chipset MTH Bug (2000)

- Error in memory translator hub
- Recall of around 1 Mio. motherboards
- \$ 253 Mio. financial loss

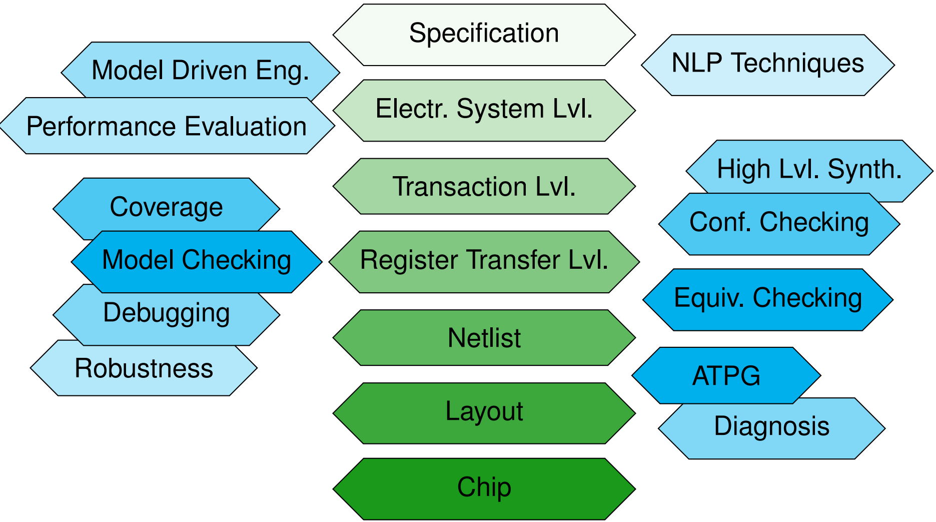
Intel i6/i7 (Skylake) Hyperthreading Bug (2017)

- Specific operating conditions cause unpredictable behavior
- Found by Linux/OCaml developers

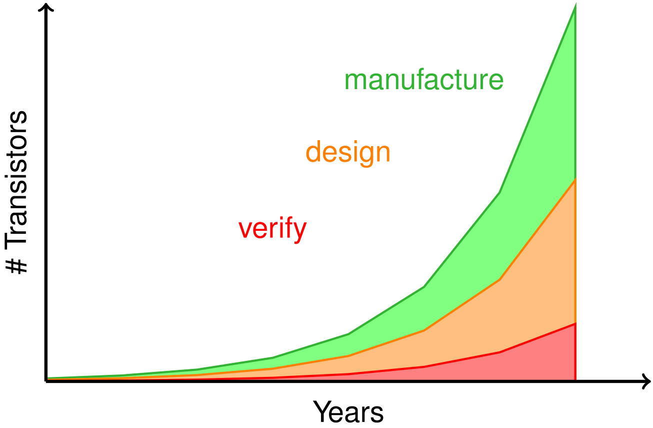
Hardware Design Flow



Hardware Verification Flow



Design Gap – Verification Gap



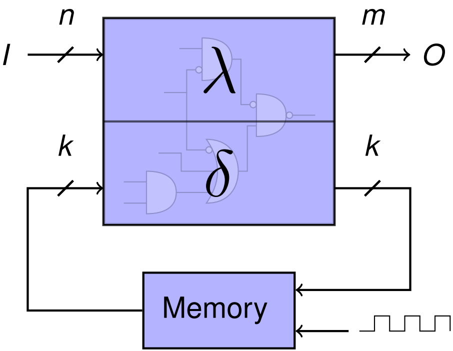
Outline

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

TELECOM ParisTech

Sequential Circuit Model



Mealy Machine:

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

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

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

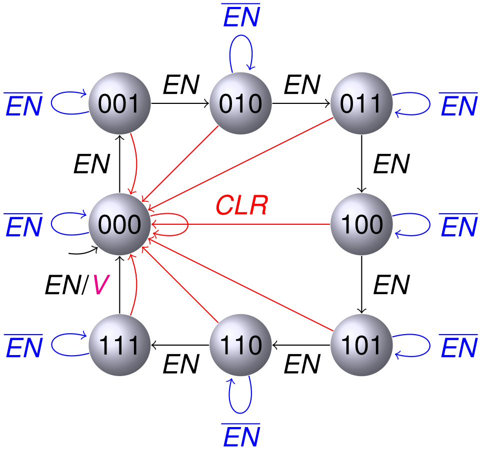
From Verilog to Mealy Machine

```verilog
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
```



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

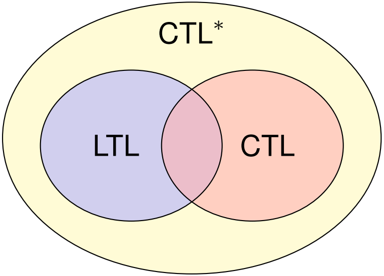
Something good will eventually happen, e.g.

"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

π_0: $000 \longrightarrow 001 \longrightarrow 010 \longrightarrow 011 \longrightarrow \cdots$

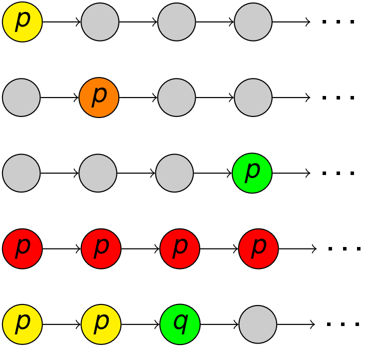
π_1: $000 \longrightarrow 001 \longrightarrow 000 \longrightarrow 001 \longrightarrow \cdots$

π_2: $000 \longrightarrow 001 \longrightarrow 010 \longrightarrow 000 \longrightarrow \cdots$

...

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	$p\ \mathbf{U}\ q$



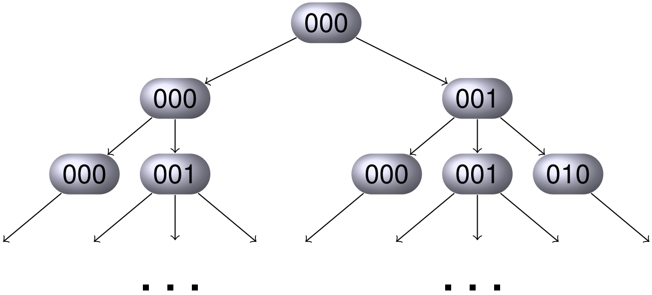
Linear Time Logic (LTL)

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

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

Branching Time View

Computation Tree

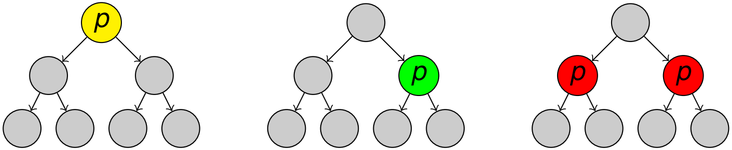


Computation Tree Logic (CTL)

Some property p holds (in the initial state)

p holds in some next state

p holds in all next states



p

EX p

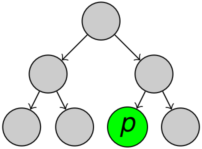
AX p

path quantifier

next operator

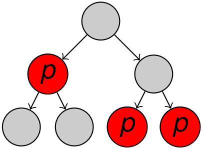
Further Modalities

p holds in some future state



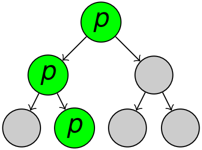
EF *p*

p holds eventually



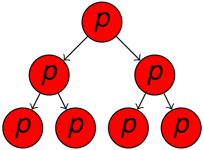
AF *p*

p holds globally on some path



EG *p*

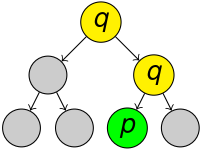
p holds globally on all paths



AG *p*

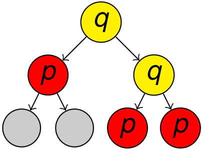
Until Modalities

On some path, q holds until p holds



$\mathbf{E}(p\ \mathbf{U}\ q)$

On all paths, q holds until p holds



$\mathbf{A}(p\ \mathbf{U}\ q)$

Computation Tree Logic (CTL)

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

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

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

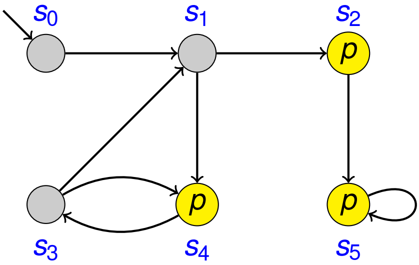
Model Checking

Given a Mealy Machine $\mathcal{M}$ and a CTL formula φ,
check if $\mathcal{M} \models \varphi$.

How do we do this?

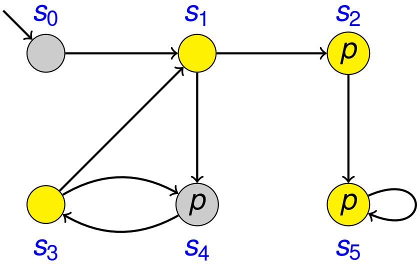
1. Compute all states in which φ holds:
   $\tau(\varphi) = \{s \in S \mid \mathcal{M}, s \models \varphi\}$
2. Check if the initial states are a subset of those states:
   $S_0 \setminus \tau(\varphi) = \varnothing$

Example



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

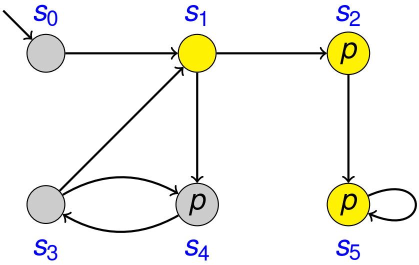
Example



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

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

Example

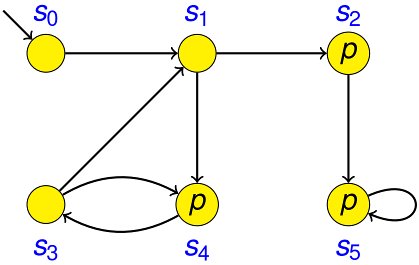


$\tau(p) = \{s_2, s_4, s_5\}$

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

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

Example



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

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

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

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

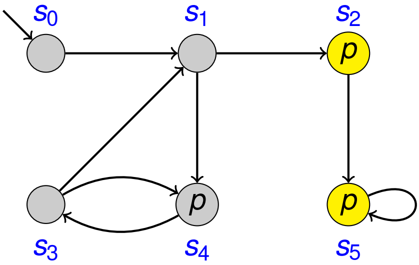
$\cup \{s_1, s_3\} \cup \{s_0\}$

Expansion rules:

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

TELECOM ParisTech

Example



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

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

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

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

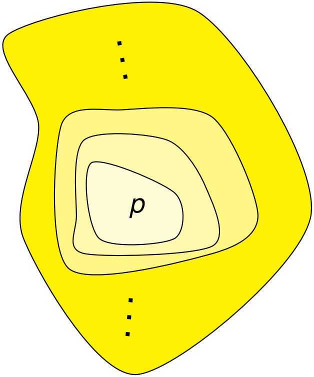
$\tau(\mathbf{AG}\ p) = \{s_2, s_4, s_5\} \cap \{s_2, s_5\}$

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

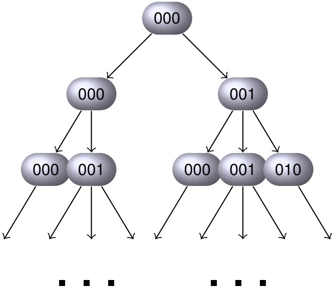


$$S_0 = p$$
$$S_1 = p \cup \mathbf{EX}\ p$$
$$S_2 = p \cup \mathbf{EX}\ p \cup \mathbf{EX}\ \mathbf{EX}\ p$$
$$\ldots$$
$$S_n = p \cup \bigcup_{i=1}^{n} \mathbf{EX}^i p = S_{n-1}$$

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

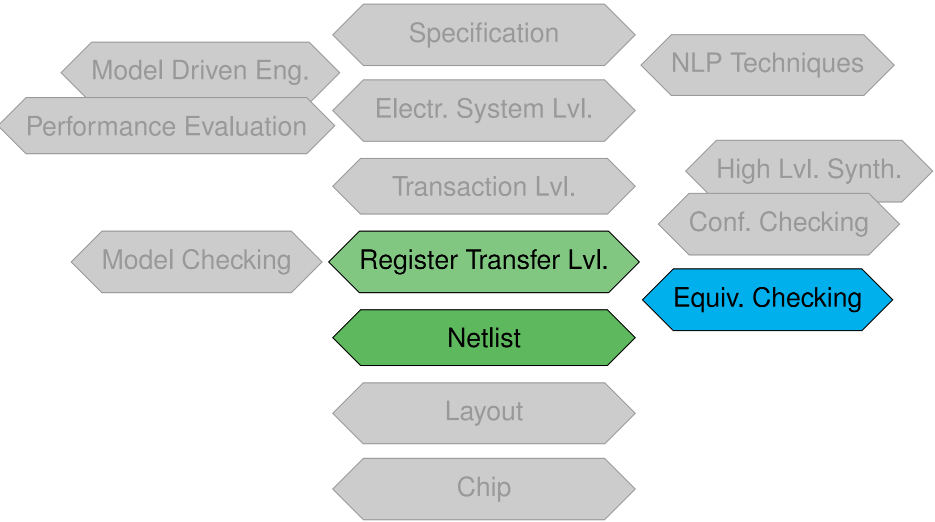
Model Checking

- 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



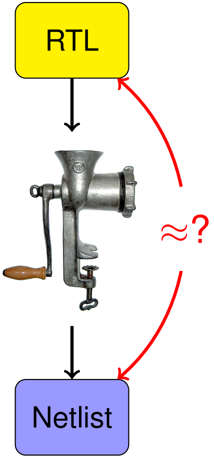
Outline

Equivalence Checking

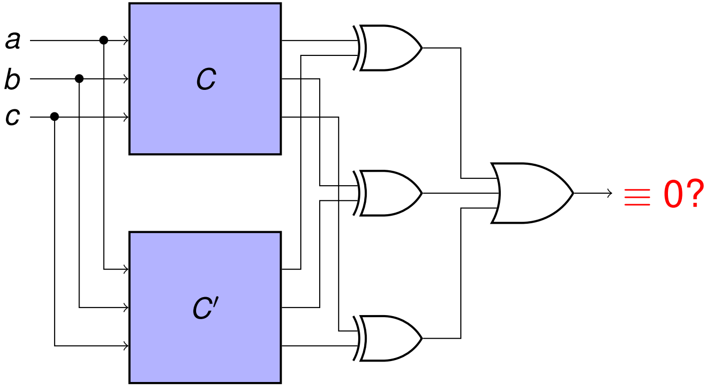


Equivalence Checking

- Hardware *synthesis* is complex
- Aggressive *optimization*
- Automatic pipelining, retiming, . . .
- Technology mapping
- Tools are closed source
- Does the netlist do what we think it does?



Miter Structure



Boolean Satisfiability (SAT)

SAT Problem

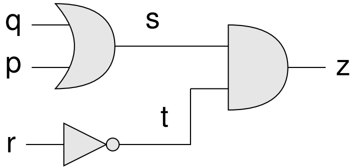
Given a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ (in conjunctive normal form), is there an assignment $X \in \{0,1\}^n$, such that $f(X) = 1$?

Conjunctive Normal Form

A Boolean formula over variables $X = \{x_0 \dots x_n\}$ is in conjunctive normal form if it is a conjunction of clauses $(l_{1,1} \vee l_{1,2} \vee \cdots \vee l_{1,m_0}) \wedge \cdots \wedge (l_{k,1} \vee \cdots \vee l_{k,m_k})$. A clause is a disjunction of literals $l = x_i$ or $l = \neg x_i$ for some $x_i \in X$.

- NP-complete problem [Cook, 1971]

Tseitin Transformation



$$z \leftrightarrow (p \vee q) \wedge \neg r$$

$$s \leftrightarrow p \vee q$$
$$t \leftrightarrow \neg r$$
$$z \leftrightarrow s \wedge t$$

$$\begin{aligned}
s \leftrightarrow p \vee q &\equiv (s \rightarrow p \vee q) \wedge (p \vee q \rightarrow s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg(p \vee q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg p \vee s) \wedge (\neg q \vee s)
\end{aligned}$$

$$t \leftrightarrow \neg r \equiv (\neg t \vee \neg r) \wedge (t \vee r)$$

$$z \leftrightarrow s \wedge t \equiv (\neg s \vee \neg t \vee z) \wedge (s \vee \neg z) \wedge (t \vee \neg z)$$

How it Looks Like in Practice...

```
c  example circuit
c
p cnf 6 8
-4 1 2 0
-1 4 0
-2 4 0
-5 -3 0
5 3 0
-4 -3 6 0
4 -6 0
5 -6 0
```

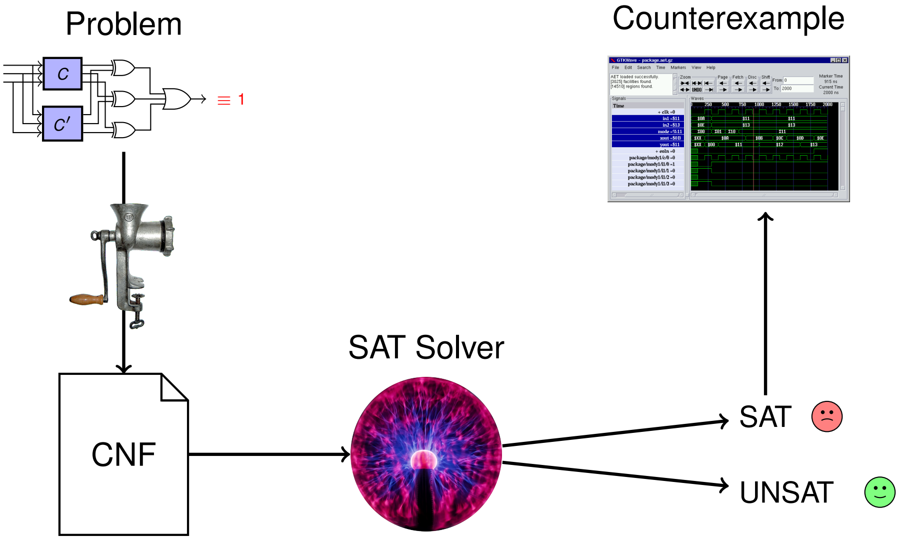
$(\neg s \vee p \vee q) \wedge$

$(\neg p \vee s) \wedge$

$(\neg q \vee s) \wedge$

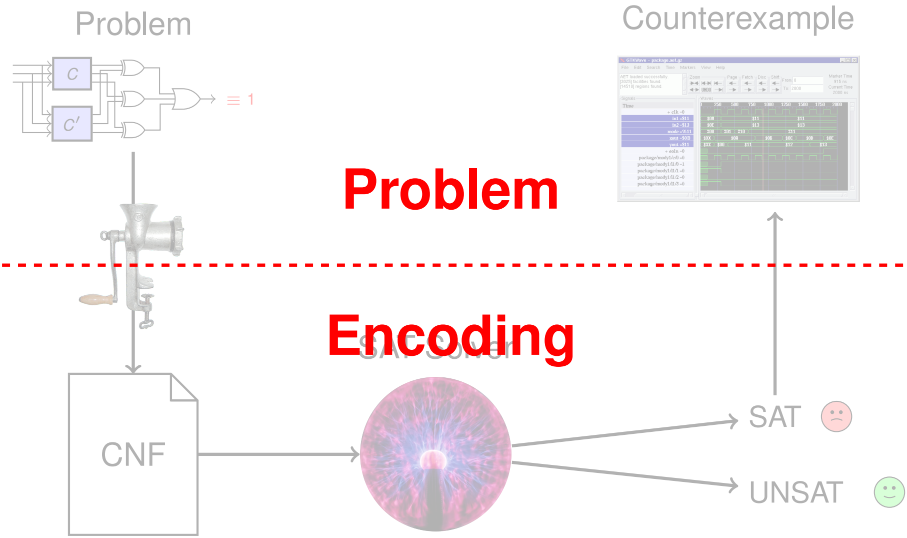
$(\neg t \vee \neg r) \ \wedge \dots$

SAT-Based Equivalence Checking



TELECOM ParisTech

SAT-Based Equivalence Checking



DPLL Algorithm

- Davis-Putnam-Logemann-Loveland [Davis et al., 1962]

DPLL

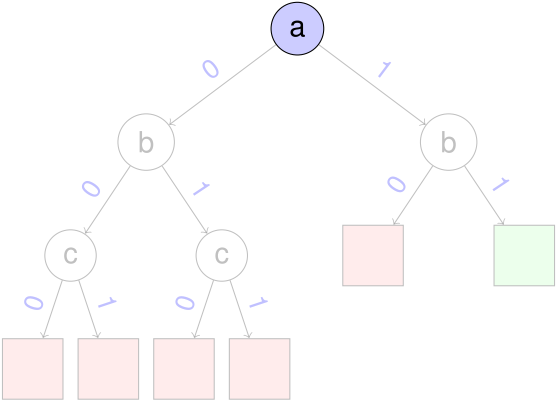
Data: Set of clauses $\mathcal{C}$

DPLL($\mathcal{C}$) **begin**

  if *all clauses satisfied* **then**
    return *SAT*
  if $\mathcal{C}$ *contains empty clause* **then**
    return *UNSAT*
  Propagate();
  $\ell \leftarrow$ *ChooseLiteral*();
  return *DPLL*($\mathcal{C} \wedge \ell$) *or* *DPLL*($\mathcal{C} \wedge \neg\ell$)

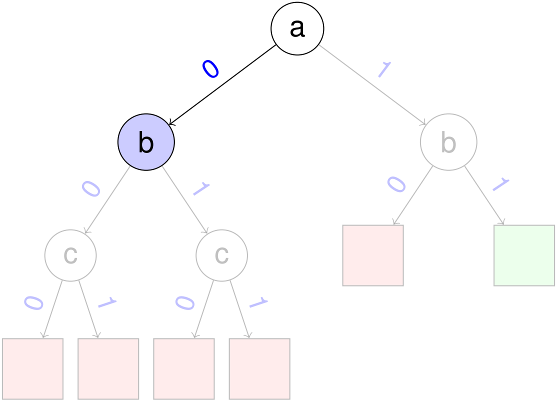
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



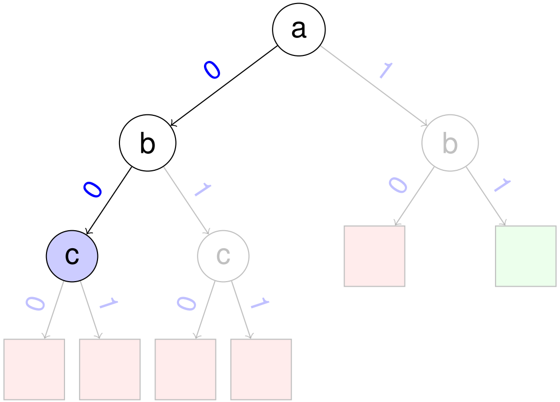
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



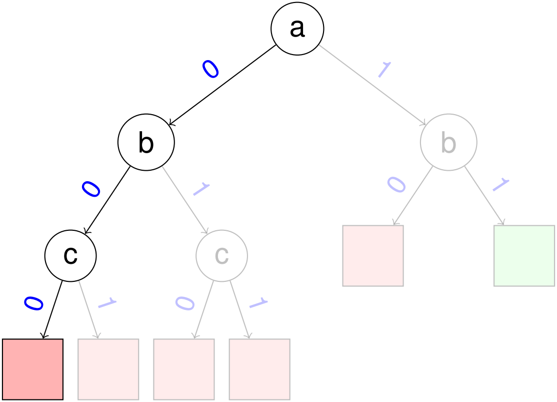
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



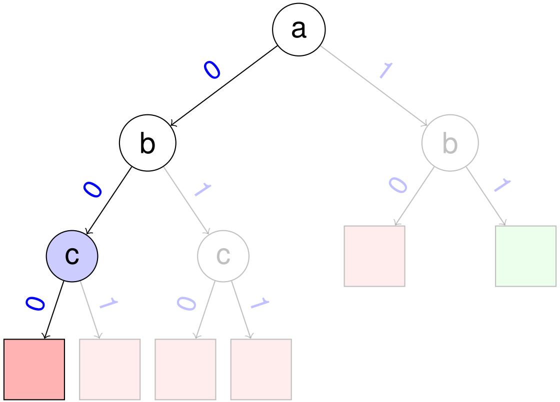
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



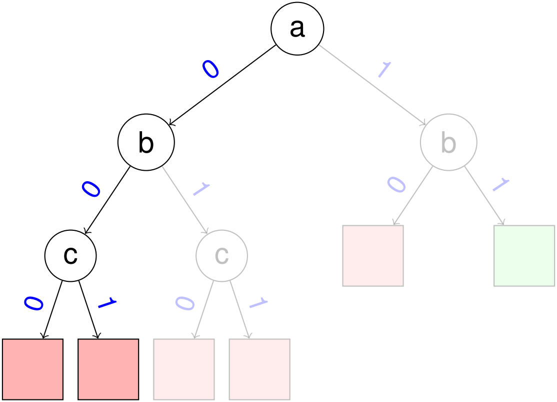
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



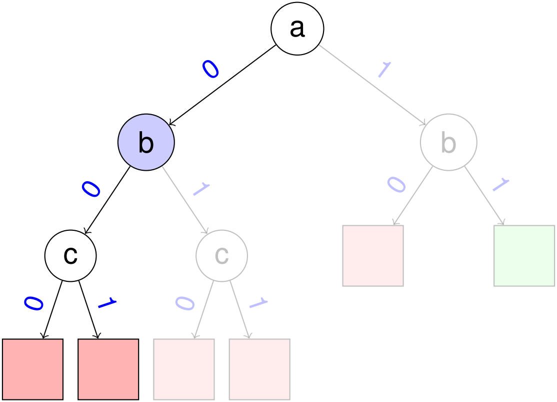
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



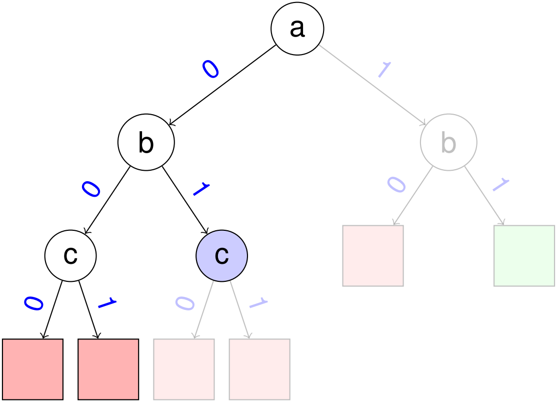
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



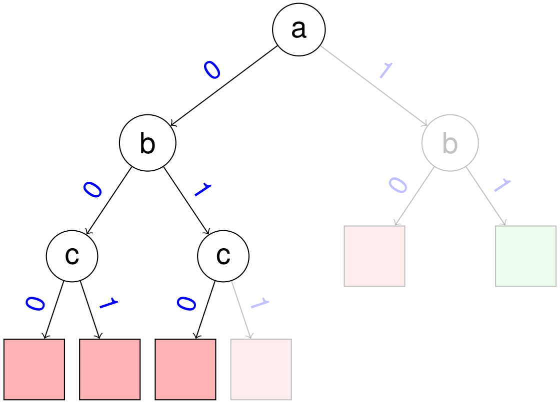
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



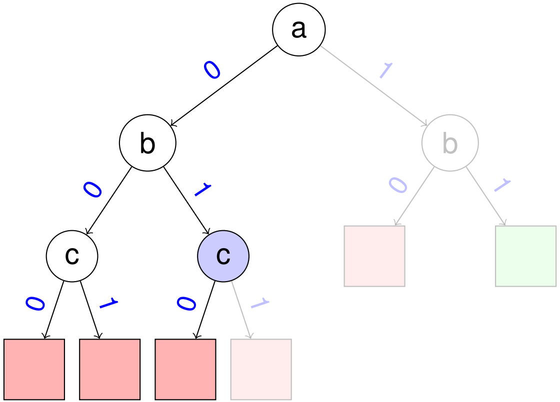
SAT Solving in a Nutshell

$(\neg a \lor b \lor c)$
$(a \lor c \lor d)$
$(a \lor c \lor \neg d)$
$(a \lor \neg c \lor d)$
$(a \lor \neg c \lor \neg d)$
$(\neg b \lor \neg c \lor d)$
$(\neg a \lor b \lor \neg c)$
$(\neg a \lor \neg b \lor c)$



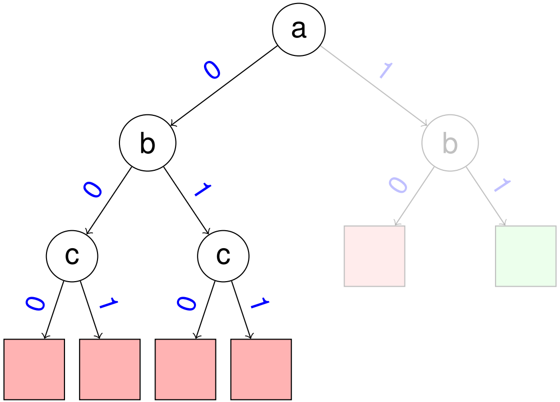
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



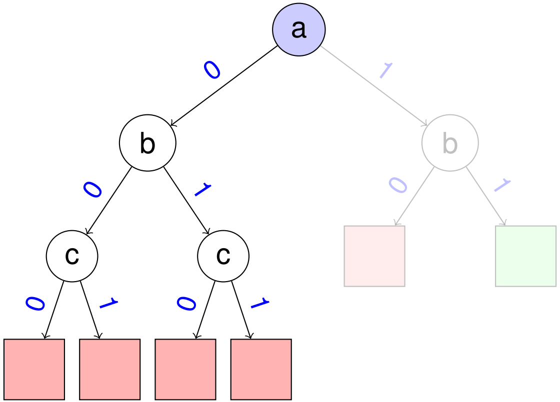
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



SAT Solving in a Nutshell

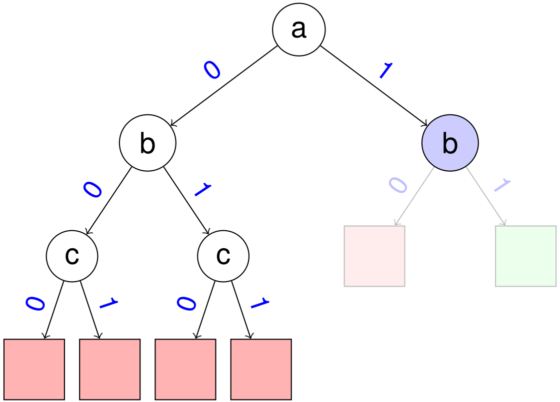
$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



TELECOM ParisTech

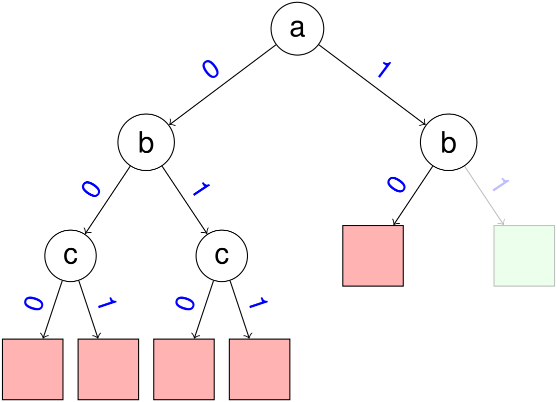
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



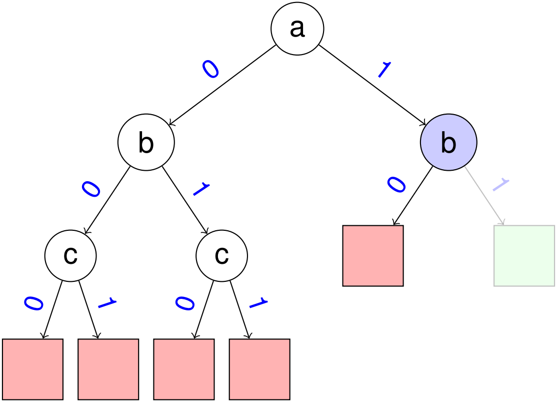
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$

$(a \vee c \vee d)$

$(a \vee c \vee \neg d)$

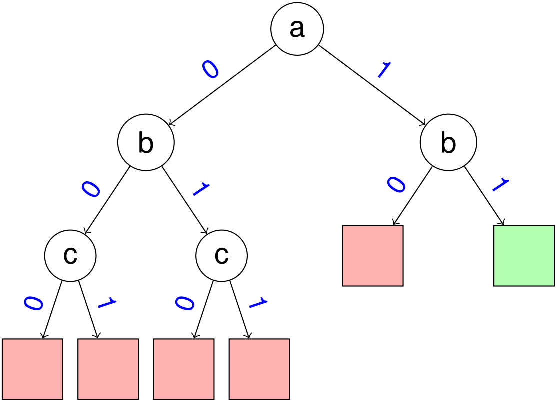
$(a \vee \neg c \vee d)$

$(a \vee \neg c \vee \neg d)$

$(\neg b \vee \neg c \vee d)$

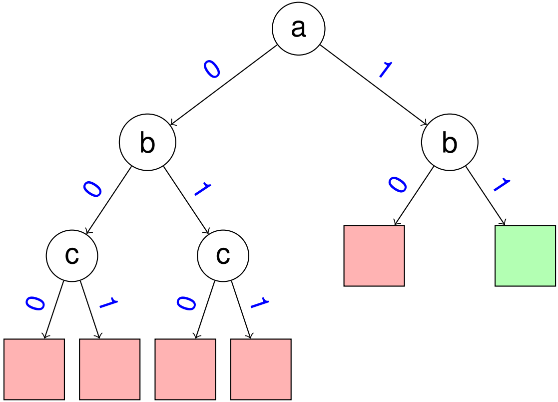
$(\neg a \vee b \vee \neg c)$

$(\neg a \vee \neg b \vee c)$



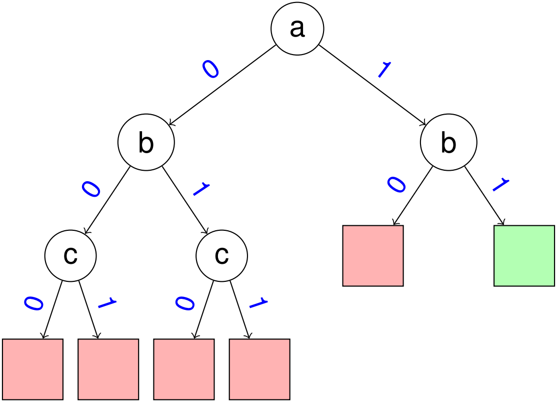
SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



Advancements in SAT Solving

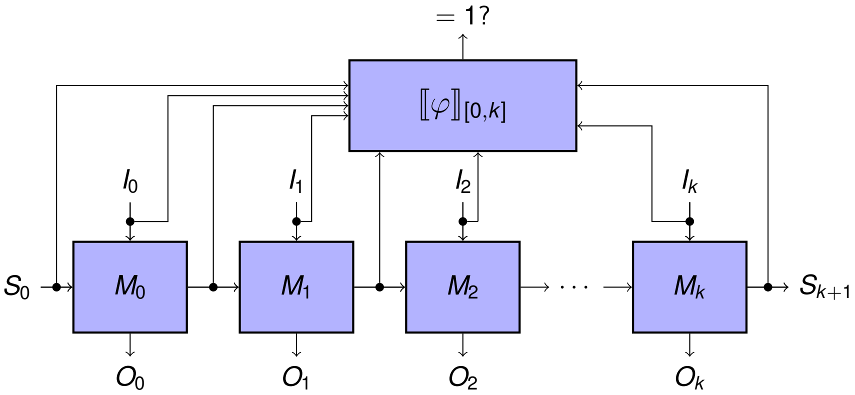
- **1962:** DPLL backtracking algorithm [Davis et al., 1962]
- **1996:** Conflict learning [Silva and Sakallah, 1996]
- **2001:** Local search, decision heuristics, engineering . . .

- Modern solvers handle millions of variables and clauses
- Popular solvers:
  - MiniSAT
  - Glucose
  - Lingeling
- Extensions of SAT:
  - Satisfiability Modulo Theories (SMT)
  - Quantified Boolean Formulae (QBF)
  - Max-SAT

Summary Equivalence Checking and SAT

- Verifying correctness of hardware synthesis
- Separation of verification problem and decision engine
- Many more applications in hardware & software verification
- Convenient and standardized APIs for ease of use
- Active field of research

Bounded Model Checking



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.

Cook, S. (1971).
The complexity of theorem proving procedures.
In *3. ACM Symposium on Theory of Computing*, pages 151–158.

Davis, M., Logemann, G., and Loveland, D. (1962).
A machine program for theorem-proving.
Commun. ACM, 5(7):394–397.

References II

Silva, J. a. P. M. and Sakallah, K. A. (1996).
Grasp – a new search algorithm for satisfiability.
In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA. IEEE Computer Society.