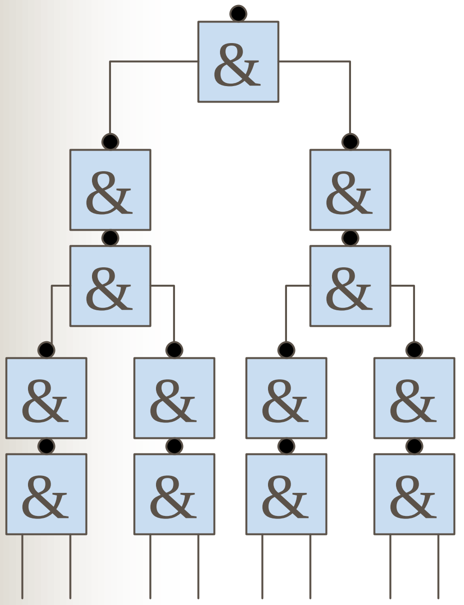
Formal background and algorithms

Formal Methods

- Goal: complete coverage
- 1st Step: precise semantics
- 2nd Step: property languages
- Tool: rigorous mathematical reasoning

Complete Coverage

- execute all possible behavior
- simulate all possible input vectors
- check abstract properties



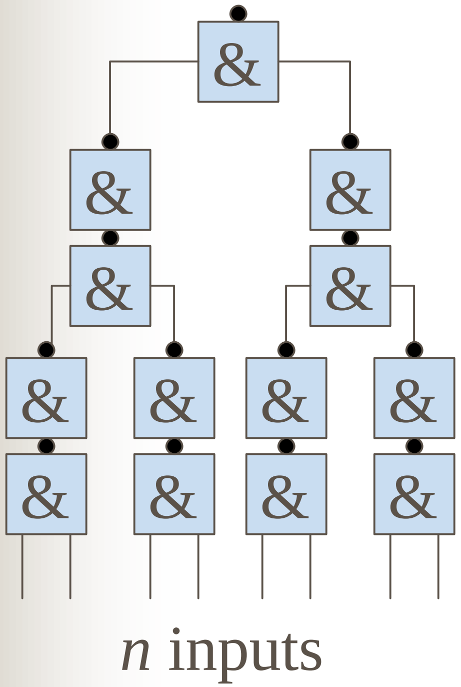
n inputs

$n+1$ instead of 2^n

symbolic test vectors	output
`0XXX XXXX`	`1`
`X0XX XXXX`	`1`
`XX0X XXXX`	`1`
`...`	`...`
`XXXX XX0X`	`1`
`XXXX XXX0`	`1`
`1111 11111`	`0`

Complete Coverage

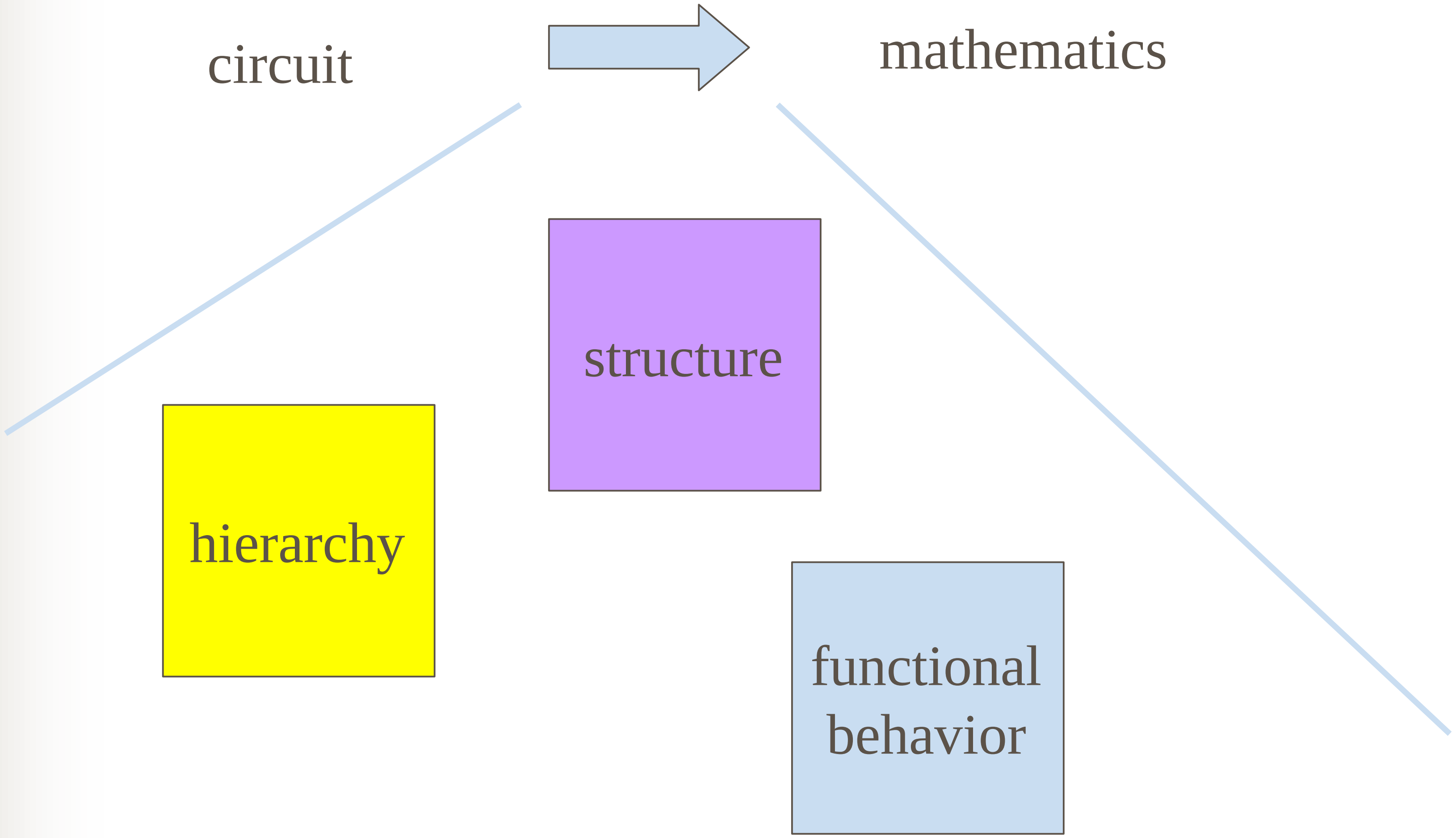
- execute all possible behavior
- simulate all possible input vectors
- check abstract properties



$n+1$ instead of 2^n

symbolic test vectors	output
`0XXX XXXX`	`1`
`X0XX XXXX`	`1`
`XX0X XXXX`	`1`
`...`	`...`
`XXXX XX0X`	`1`
`XXXX XXX0`	`1`
`1111 11111`	`0`

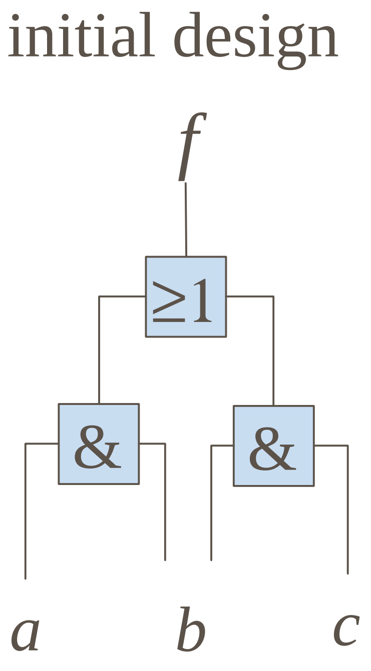
Precise Semantics

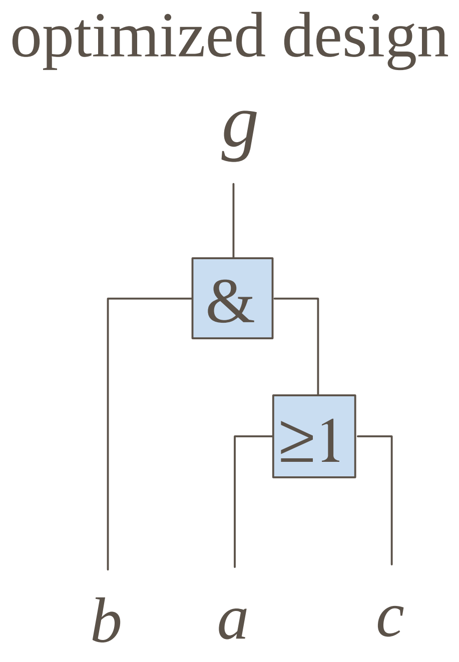


Mathematical Representation of Combinational Circuits

- structural representation
  net lists, equations, signed-and-graphs
  propositional formulae, CNF
  terms (first order), lambda-calculus (higher order)
- semantic representation
  karnaugh maps, function tables
  decision diagrams (BDD, BMD, ADD, ...)
  term rewriting systems (no other canonical)

Equivalence Checking of Combinational Circuits

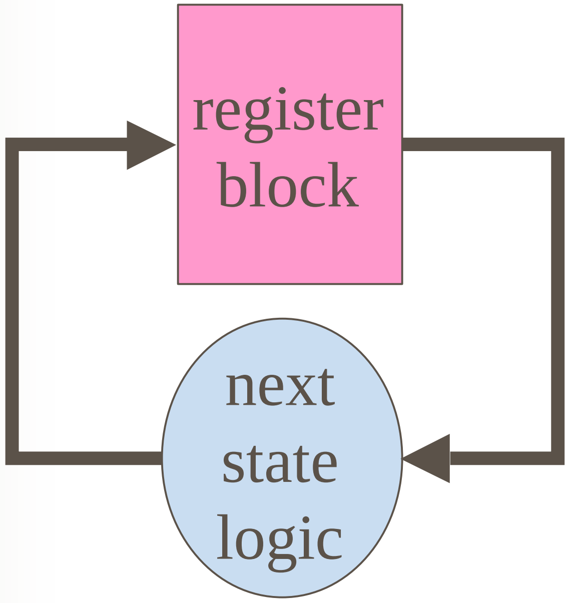


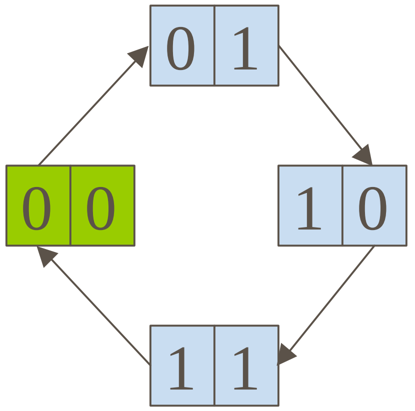


$$\forall\, a,b,c. \quad f\,(a,b,c\,) = g\,(a,b,c\,)$$

Sequential Properties

sequential circuit → finite automata

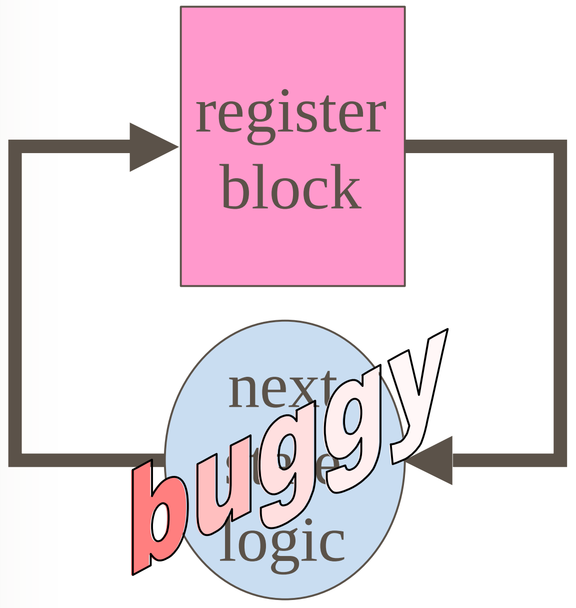


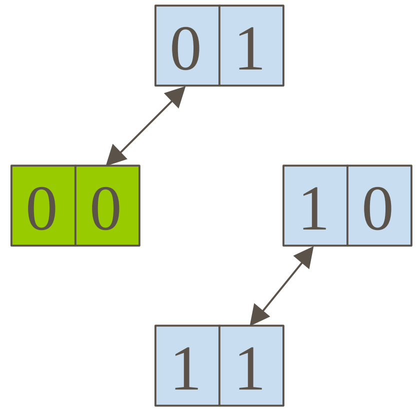


safety property: can 1 1 be reached?

Sequential Properties

sequential circuit → finite automata





liveness property: will 1 1 always be reached?

Structural Properties

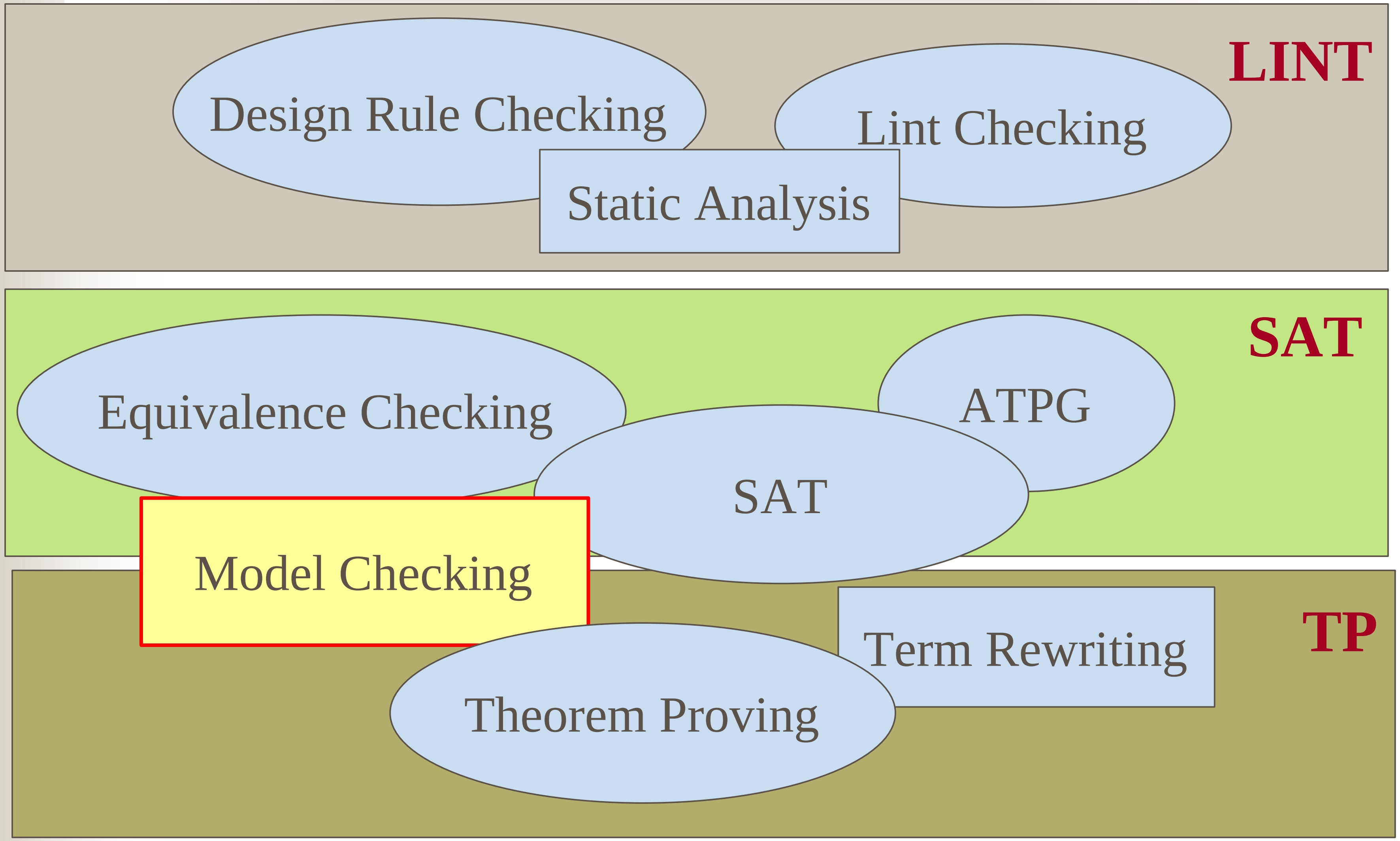
```
library IEEE;
use IEEE.std_logic_1164.all;
entity ADDER is
  generic (N : in integer := 32);
  port(
    A, B : in  std_logic_vector(N-1 downto 0);
    CI   : in  std_logic;
    S    : out std_logic_vector(N-1 downto 0);
    COUT : out std_logic);
end ADDER;
architecture RTL of ADDER is
...
end RTL;
```

generic parameter

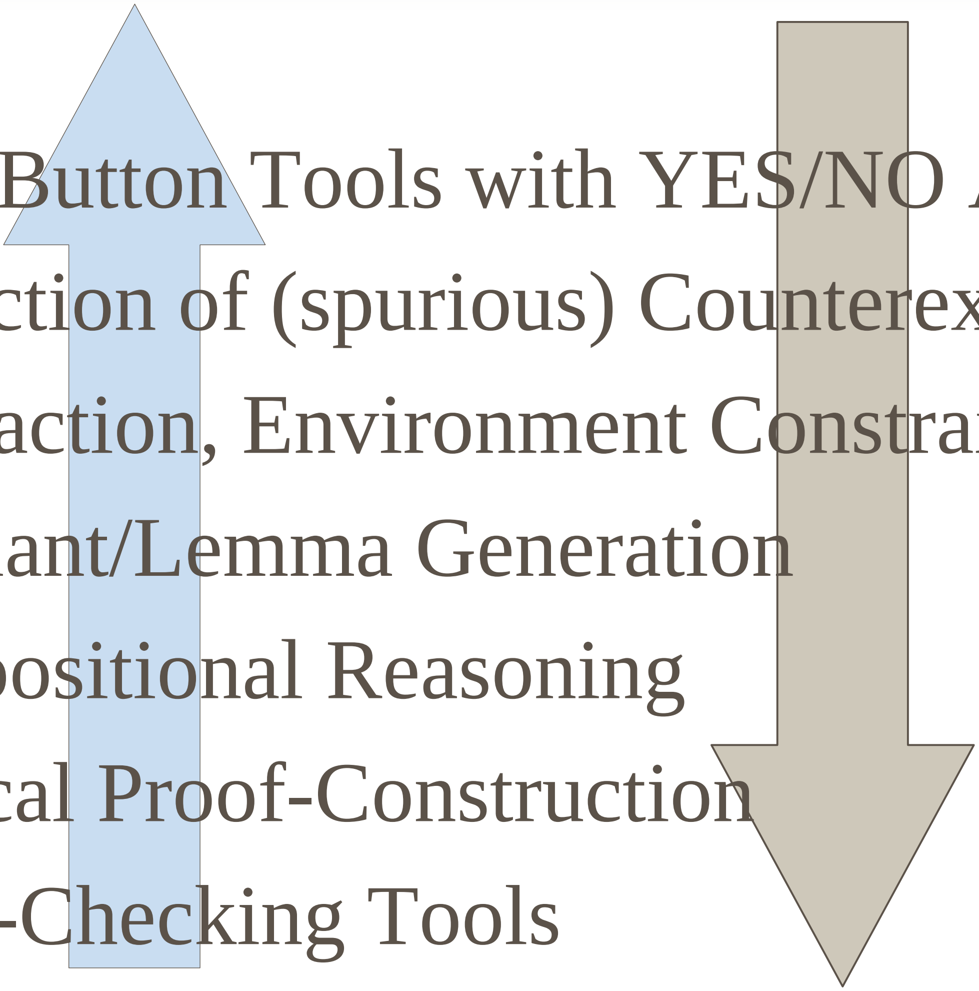
first order formula

$$\forall\, n,a,b,s,c.\quad \mathrm{RTL}(n,a,b,0,s,c) \Rightarrow a+b = s + (c \ll n)$$

Reasoning Techniques about Circuits

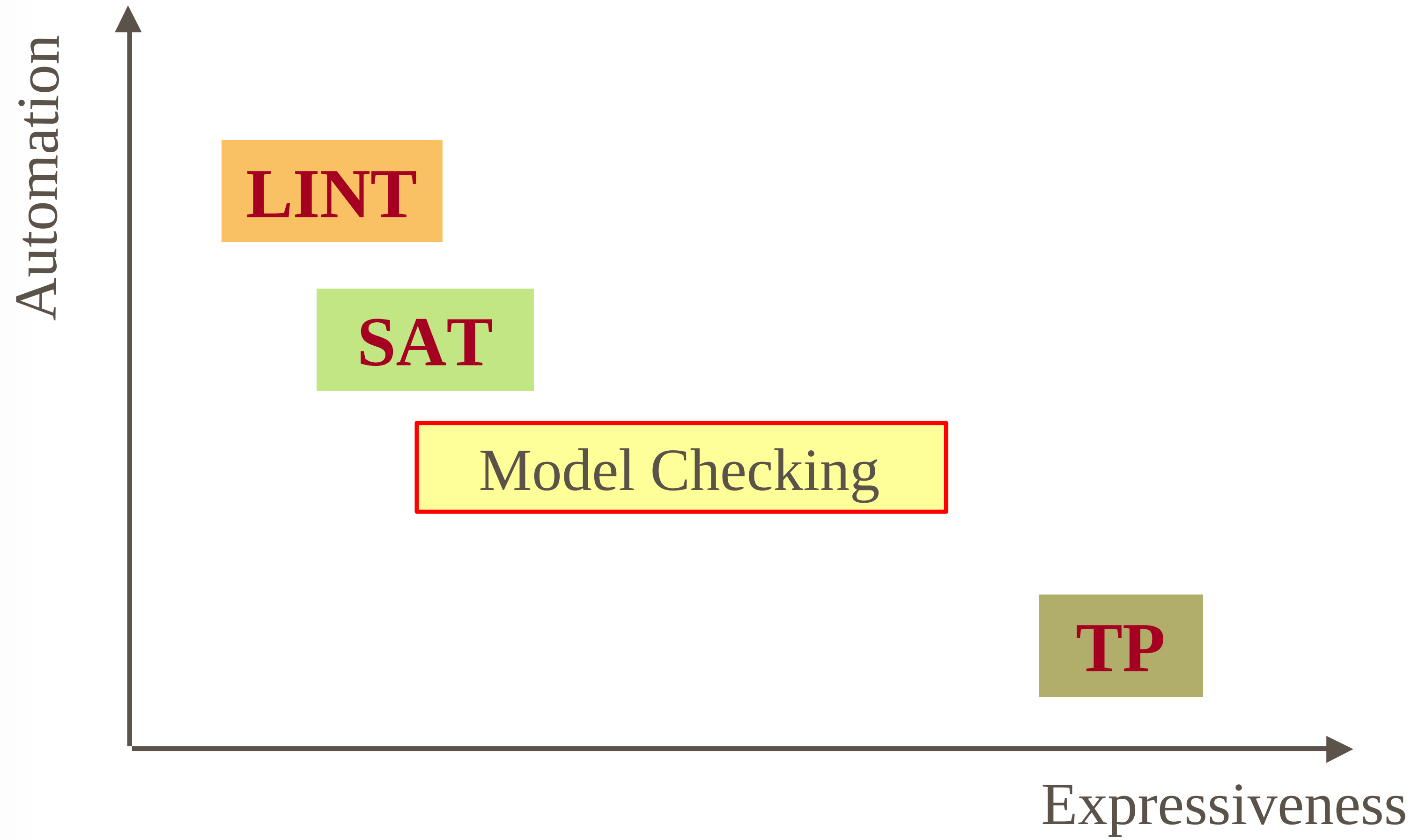


Automation versus Interaction



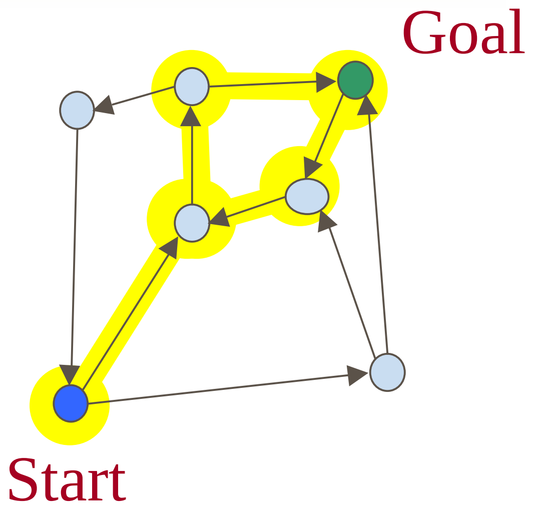
- Push-Button Tools with YES/NO Answer
- Inspection of (spurious) Counterexamples
- Abstraction, Environment Constraints
- Invariant/Lemma Generation
- Compositional Reasoning
- Tactical Proof-Construction
- Proof-Checking Tools

Automation versus Expressiveness



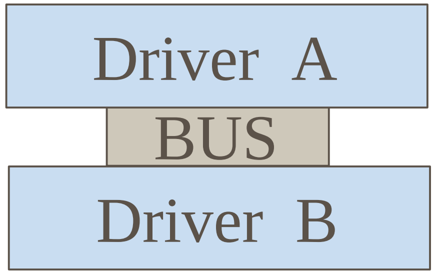
Model Checking

- State Space Exploration
- Breadth First Search (BFS)
- Depth First Search (DFS)
- Look for Traces as well
- Find States with certain Properties
- Find Loop on which certain Properties hold
- Specify Properties with Temporal Logic



Typical Properties (Safety)

- No Bus Contention
  (drivers never access the
  bus simoultaneously)



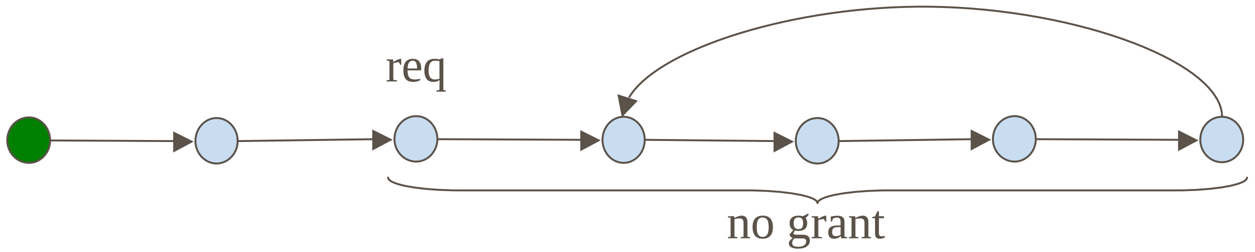
- Some vector is a one-hot-encoding
- No grant without a previous request
- Generally: Verilog $assert() does not fail

Search for violating states starting from the initial states (after reset)

Typical Properties (Liveness)

- requests will finally be granted
- no deadlocks or livelocks
- For instance:
  root assignment in FireWire™ terminates

Search for unsuccessful looping traces starting from the initial states (after reset)



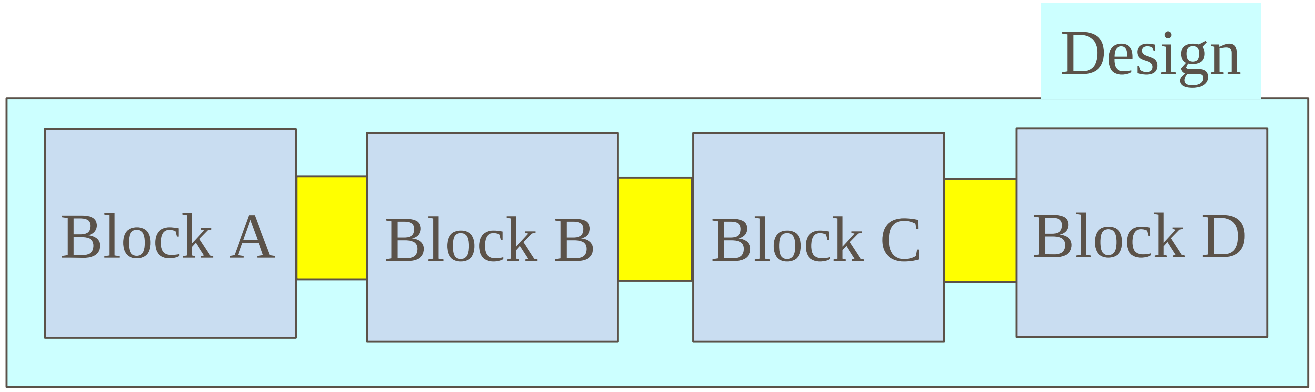
Temporal Logic

- LTL, CTL, μ-calculus
  - Mathematical language for describing sequential properties
  - Formulation of complex properties require skill
- Templates
  - Verilog extensions
  - Open Verification Library (OVL)
  - Look at templates, learn from others, change parts (if you are not a mathematician)

LTL

- Linear Temporal Logic fixes one execution trace, relates properties along this traces
- Safety: **G** *exclusive* (globally)
- Liveness: **F** *initialized* (finally)
- Nesting: $\mathbf{G}\ (\mathit{request} \rightarrow \mathbf{F}\ \mathit{grant})$
- easy to comprehend, compositional (but harder to check)

State Explosion Problem



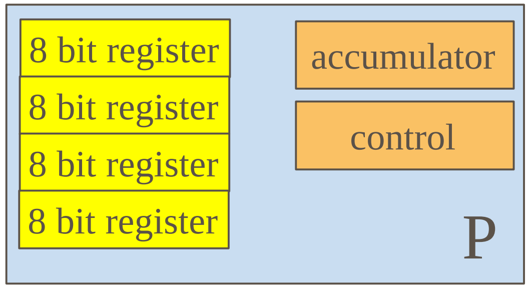
$|A|$ defined as number of flip-flops of Block A

$\|A\|$ defined as number of states of Block A

$$|Design| = |A| + |B| + |C| + |D|$$

$$\|Design\| = \|A\| \cdot \|B\| \cdot \|C\| \cdot \|D\|$$

State Explosion Problem Example



8 bit processor
4 registers, 8 bit wide
16 bit accumulator, control

$$\|P\| \approx 2^8 \cdot 2^8 \cdot 2^8 \cdot 2^8 \cdot 2^{16} \cdot 2^{16} = 2^{4\cdot 8+2\cdot 16} = 2^{64}$$

State space grows exponentially with the number of flip-flops

Explicit Model Checking

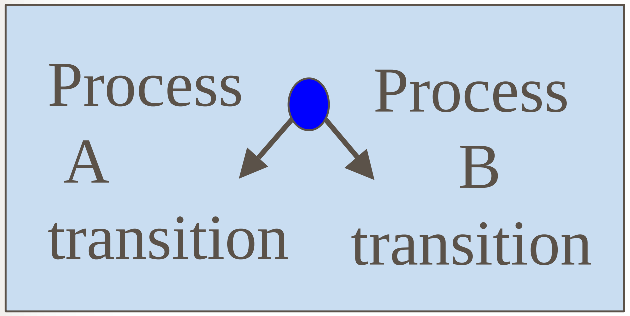
- Traverse state space in DFS
  (because the search stack needs less space)
- Save reached states explicitely in hash table
- Size of hash table $||model|| \cdot |model|$
- Limit: several millions of states (= $||model||$)
- Typical Application: interacting state machines

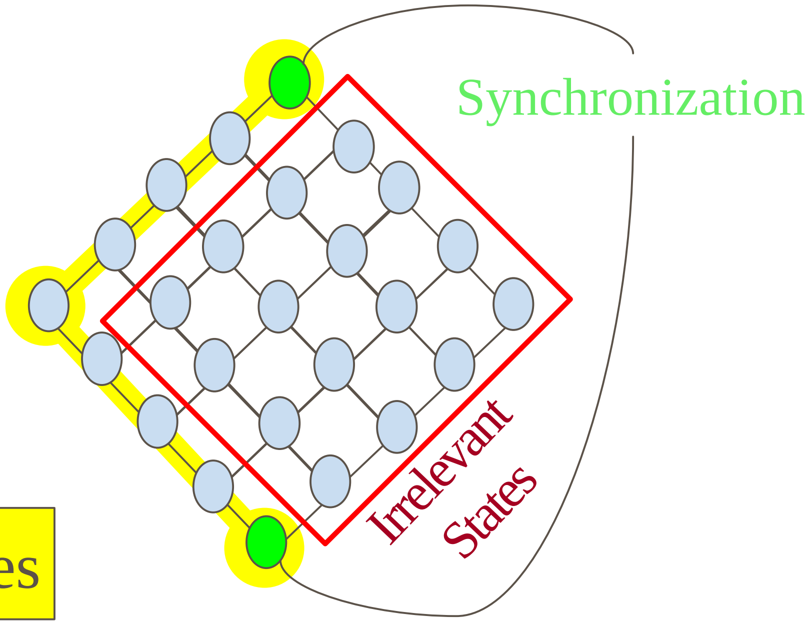
Small Designs with small number of flip-flops (<80)
and small number of primary inputs (<20)

Otherwise usually the fastest model checking technology

Partial Order Reduction

- Factor out independent state transitions! (typical for asynchronous communication as in Software- and Protocol-Checking)
- May result in exponential reduction in the number of states





One Relevant Trace of States

Explicit Model Checking

- Invented by [Clarke,Emerson]
- Academic Tools:
  - SPIN [Holzmann]
    C like input language (PROMELA)
    partial order reduction, large user base,
    applicable to protocol & software validation
  - Murphi [Dill]
    hardware oriented input language
    symmetry reduction, smaller user base

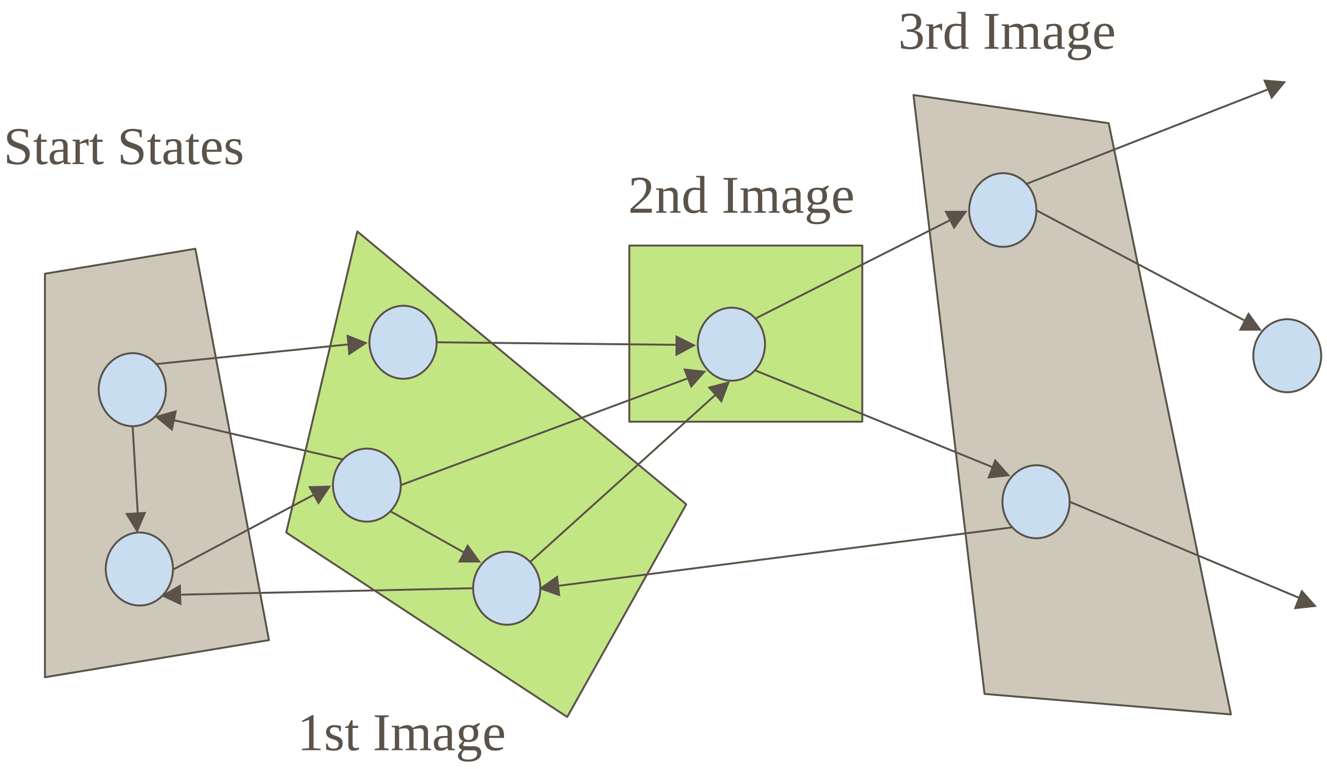
Symbolic Model Checking

- Explicit Model Checking: number of states limited ($< 10^{10}$)
- Symbolic Representation of States: potentially many more states ($> 10^{20}$)
- **However:** Complexity Theory tells us that model checking is PSPACE hard in the number of state bits (flip-flops), so probably exponential!
- **In Practice:** works for hundreds of state-bits

Symbolic Model Checking

- Set of States instead of Single States
- Represent Set of States with their Characteristic Functions
- Usually Boolean Encoding leads to Boolean Characteristic Functions
- Set Operations as Boolean Operations
- Efficient Data Structure for Boolean Functions: Binary Decision Diagrams (BDDs)

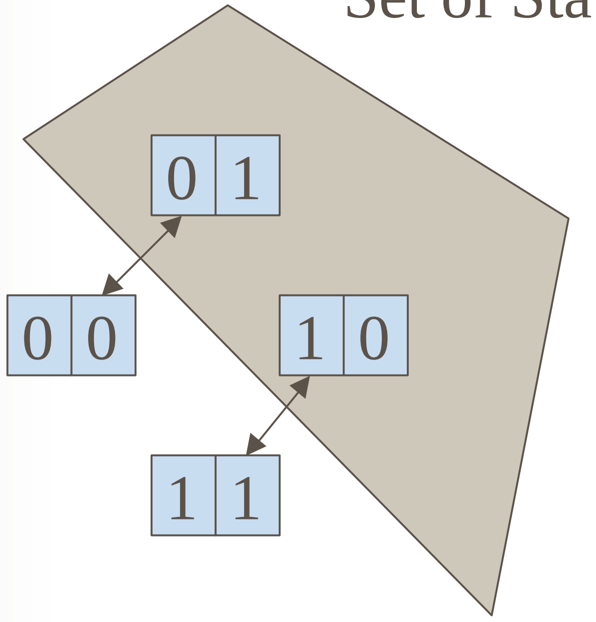
Set of States instead of Single States



Breadth First Search (BFS)

Characteristic Functions for Sets

Set of States



Function

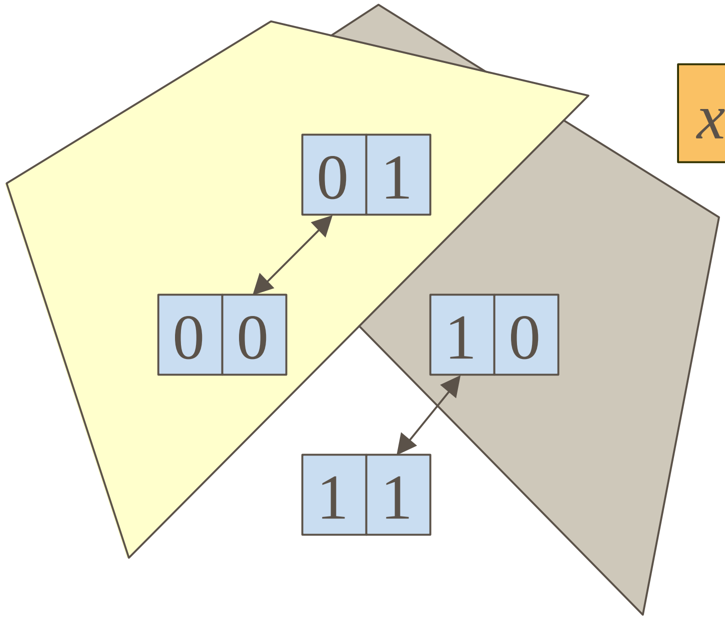
0 0	0
0 1	1
1 0	0
1 1	1

$$x \cdot (1 - y) + (1 - x) \cdot y$$

Boolean Expression

Symbolic Operations on Set of States

$(1 - x)$



$x \cdot (1 - y) + (1 - x) \cdot y$

intersection as conjunction

$(1 - x) \cdot x \cdot (1 - y) + (1 - x) \cdot y$

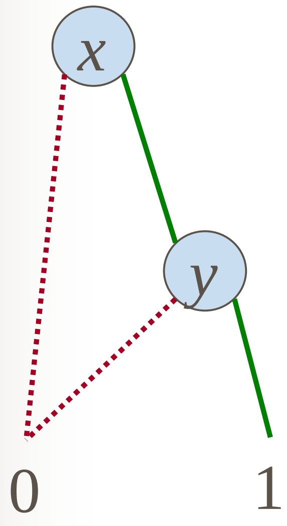
simplifies to

$(1 - x) \cdot y$

Binary Decision Diagrams

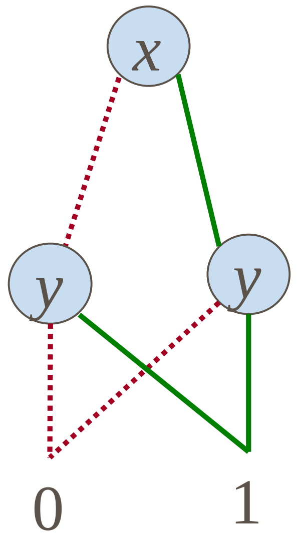
- Fixed global variable order
- Reduced Ordered BDDs by [Bryant]
- Canonical data structure for boolean functions
- Efficient (linear) boolean operations
- Good variable orders are necessary …
- … but often do not exist (e.g. multipliers)

Binary Decision Diagrams



```
if x then
  if y then 1
  else 0
else 0
```

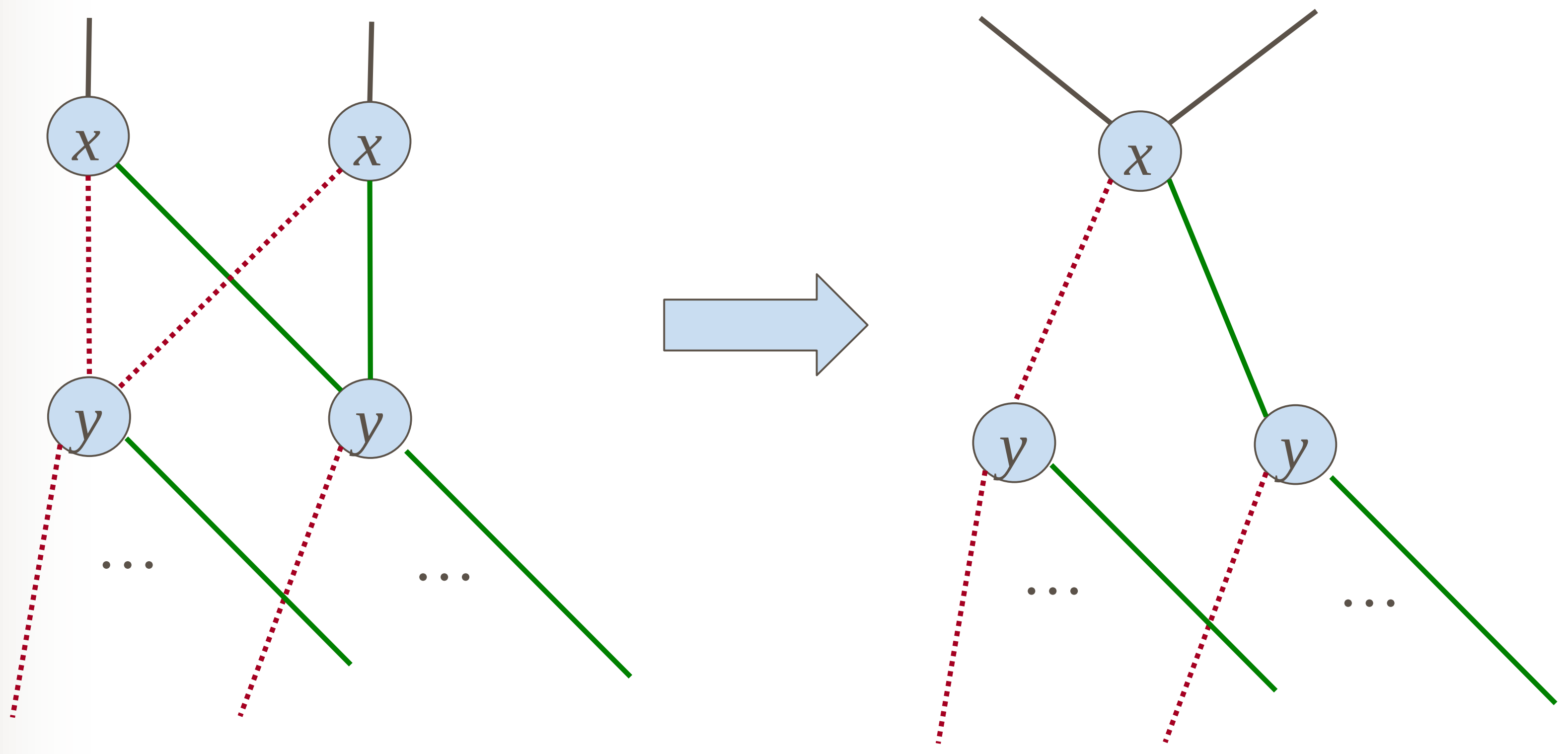
$x \wedge y$



```
if x then
  if y then 1
  else 0
else
  if y then 0
  else 1
```

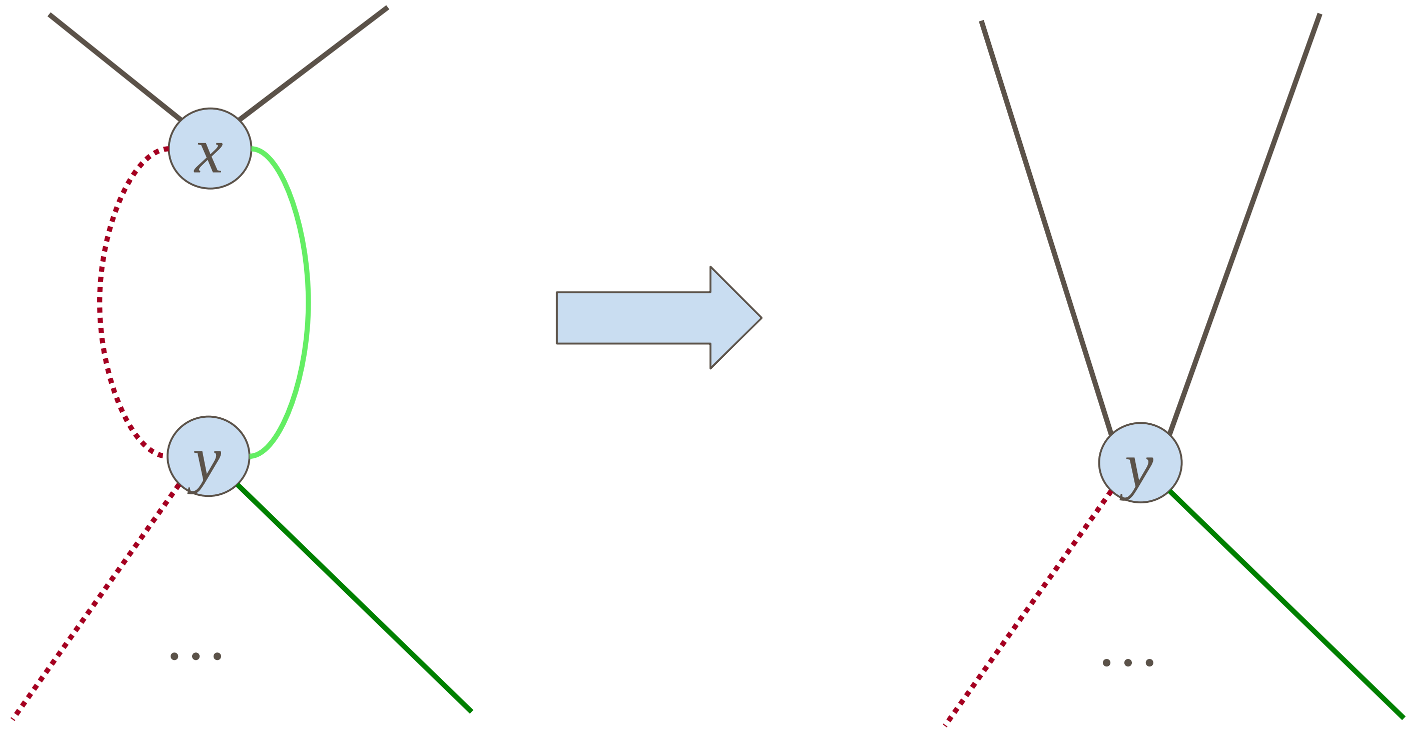
$x = y$

BDD Reduction: Merge Equivalent Nodes



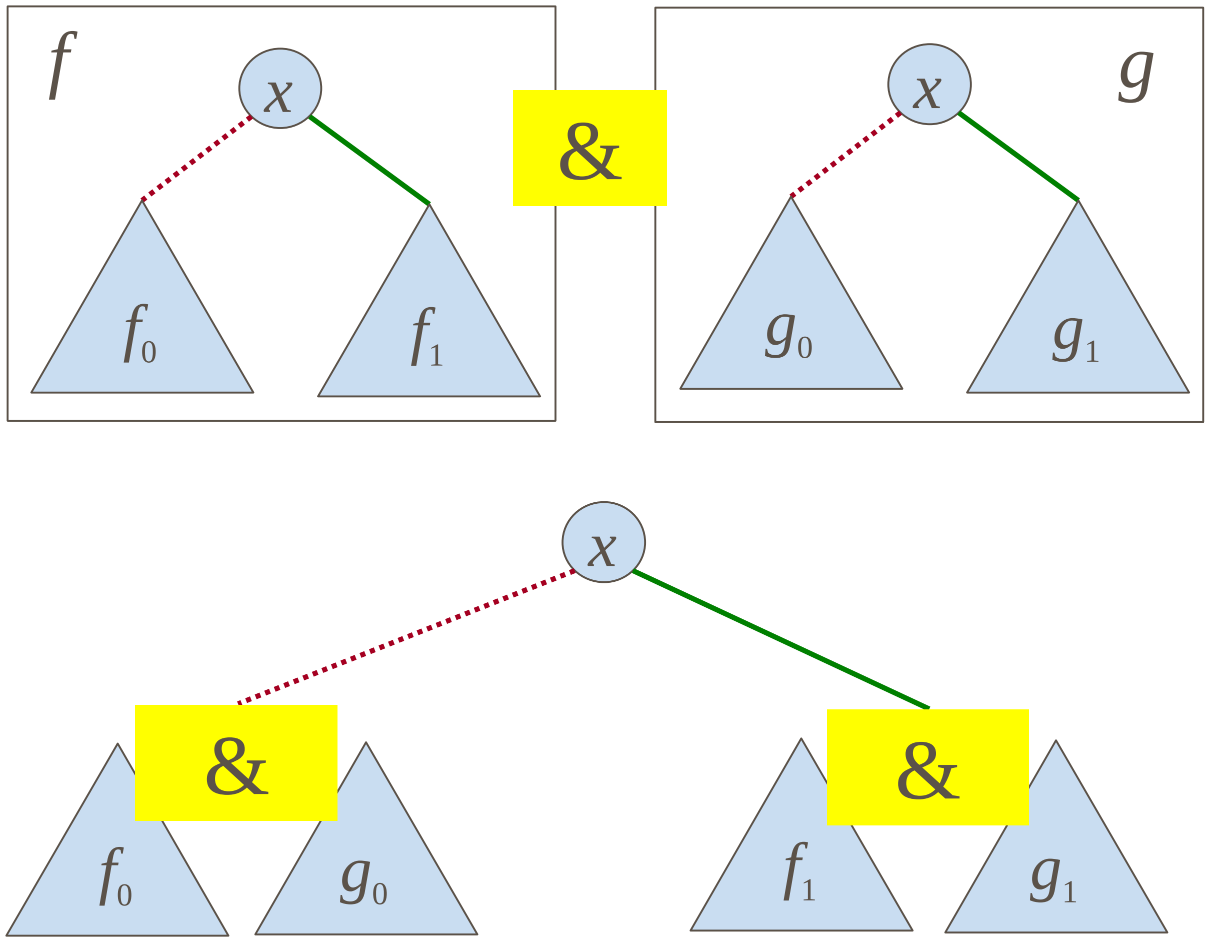
one level deep look-ahead

BDD Reduction: Eliminate Redundant Nodes



one level deep look-ahead

Recursive Step for Conjunction of BDDs



Generic BDD Operation Apply

```
cofactors(f,g)
  x = min(topvar(f),topvar(g))
  (f0,f1) = (x == topvar(f)) ? (lo(f), hi(f)) : (f,f)
  (g0,g1) = (x == topvar(g)) ? (lo(g), hi(g)) : (g,g)
  return (x,f0,f1,g0,g1)

apply(op,f,g)
  if({f,g} subset {0,1}) then return op(f,g)
  if(cached(op,f,g)) return cache(op,f,g)
  (x,f0,f1,g0,g1) = cofactors(f,g)
  l = apply(op,f0,g0)
  r = apply(op,f1,g1)
  n = node(x,l,r)
  cache(op,f,g) = n
  return n
```

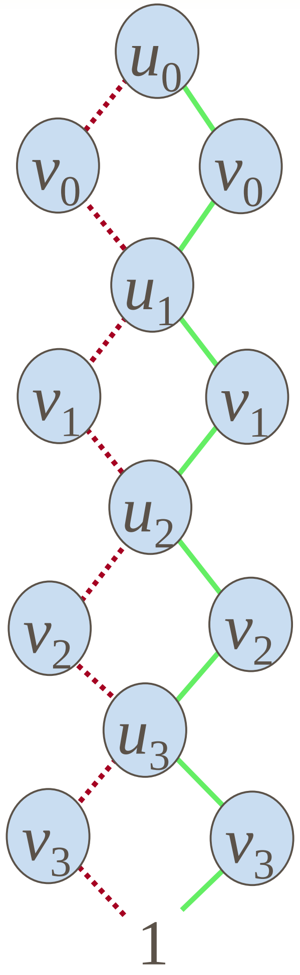
generic operator Like AND, OR, etc

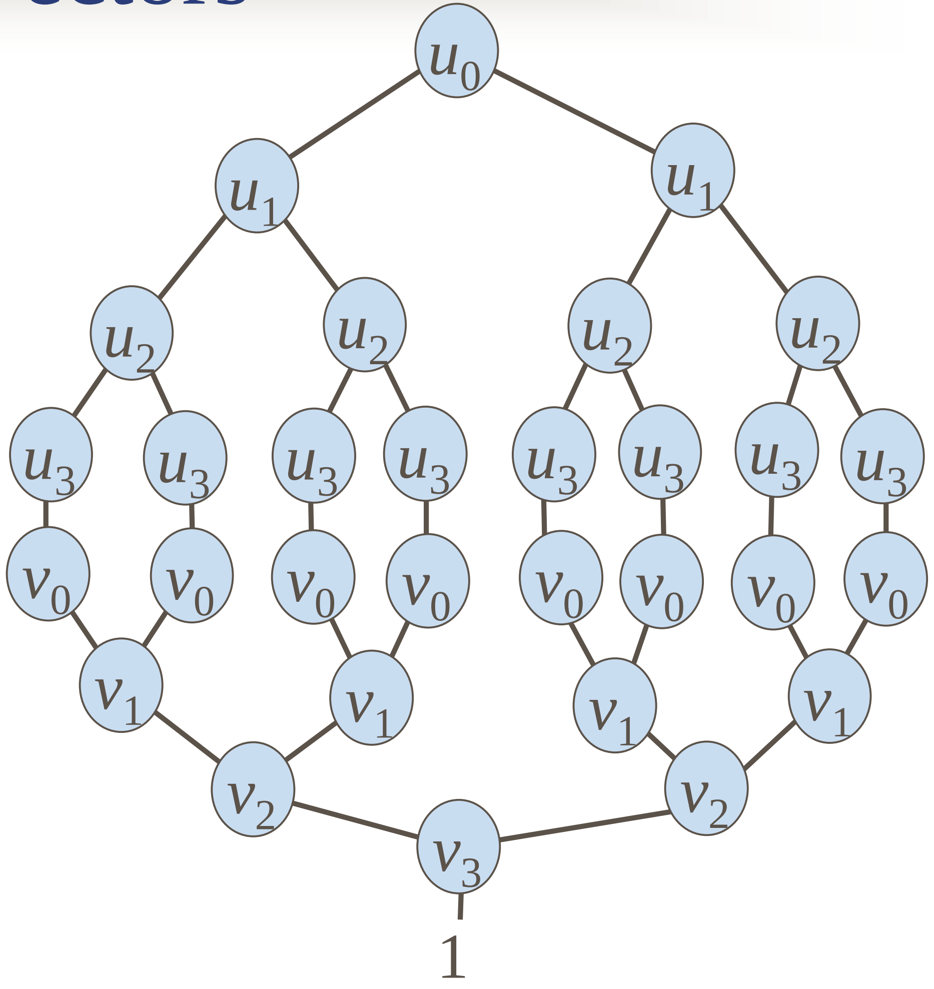
base case

recursion

reduce locally before generating new node

Variable Order for Comparison of Vectors





$(u_0, u_1, u_2, u_3) = (v_0, v_1, v_2, v_3)$

Variable Ordering in Practice

- Static Heuristics
  - Use Circuit Structure (e.g. DFS occurrence)
  - Model Checking usually does not benefit
- Dynamic Reordering [Rudell]
  - Inplace Swapping of variable levels
  - Necessary (no success without reordering)
  - Very Expensive (often dominates runtimes)
  - Fails (sometimes good orders do not exists)

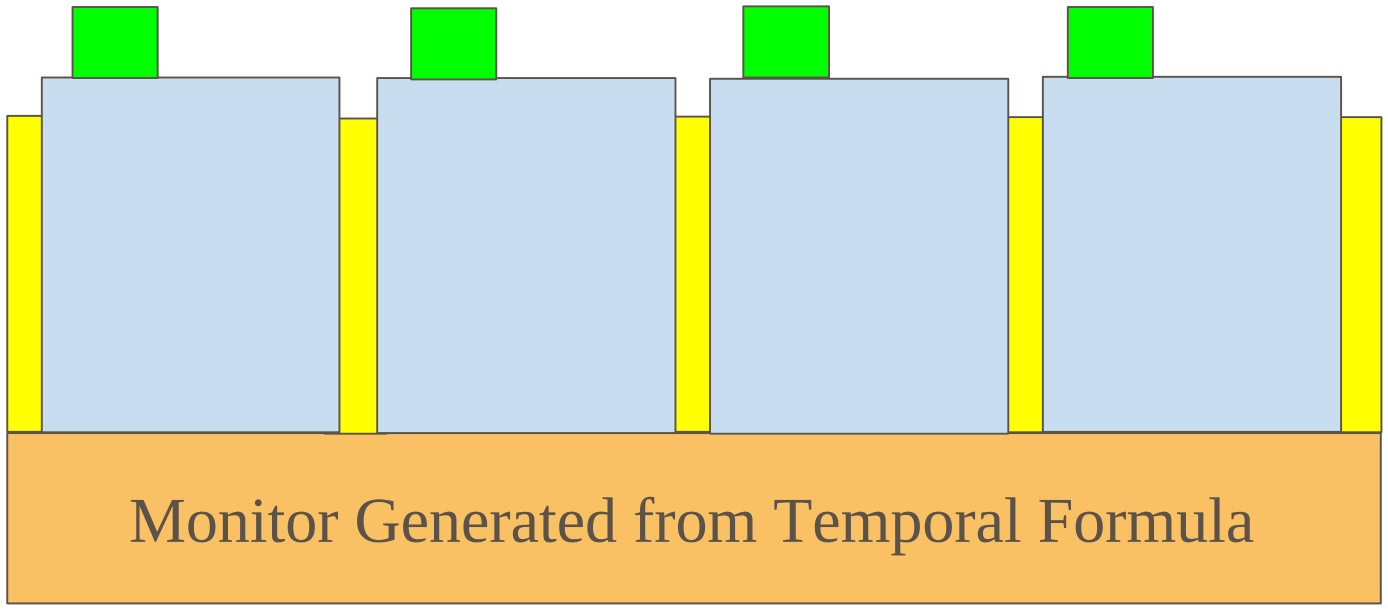
Bounded Model Checking

- Invented by [Biere,Cimatti,Clarke,Zhu]
- No Calculation of Images
- Symbolic Unrolling of Transition Relation
- Incomplete in Practice:
  Can not show absence of a bug in general
- SAT procedures for detecting reachability of a bug in a fixed number of time steps
- No Variable Ordering Problem:
  much larger designs (thousands of state bits)

Bounded Unrolling

4 copies of transition logic

4 copies of primary inputs



5 copies of state bits

Capacity of Algorithms for Checking Sequential Properties

	typical number of state bits (flip-flops)	
Explicit MC	30-100	complete
Symbolic MC (BDD)	100-500	
Bounded MC (SAT)	200-2000	incomplete
Simulation	> 10000	

(primary inputs counted as state bits)

Recipe for Applying Formal Methods

- Theorem Proving for checking algorithms
- Equivalence Checking for refinements
- Model Checking:
  - Check protocols and complex interacting state machines in high-level design
  - Check sequential properties on RTL-level
    - Simulation
    - Explicit Model Checking
    - Bounded Model Checking
    - BDD based Model Checking

Commercial Model Checkers

- 0-in
- Avanti
- Averant
- Cadence
- Innologic
- Real Intent
- Synopsys
- Verplex

…