Introduction to Electronic Design Automation

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Spring 2012

Formal Verification

Part of the slides are by courtesy of Prof. Y.-W. Chang, S.-Y. Huang, and A. Kuehlmann

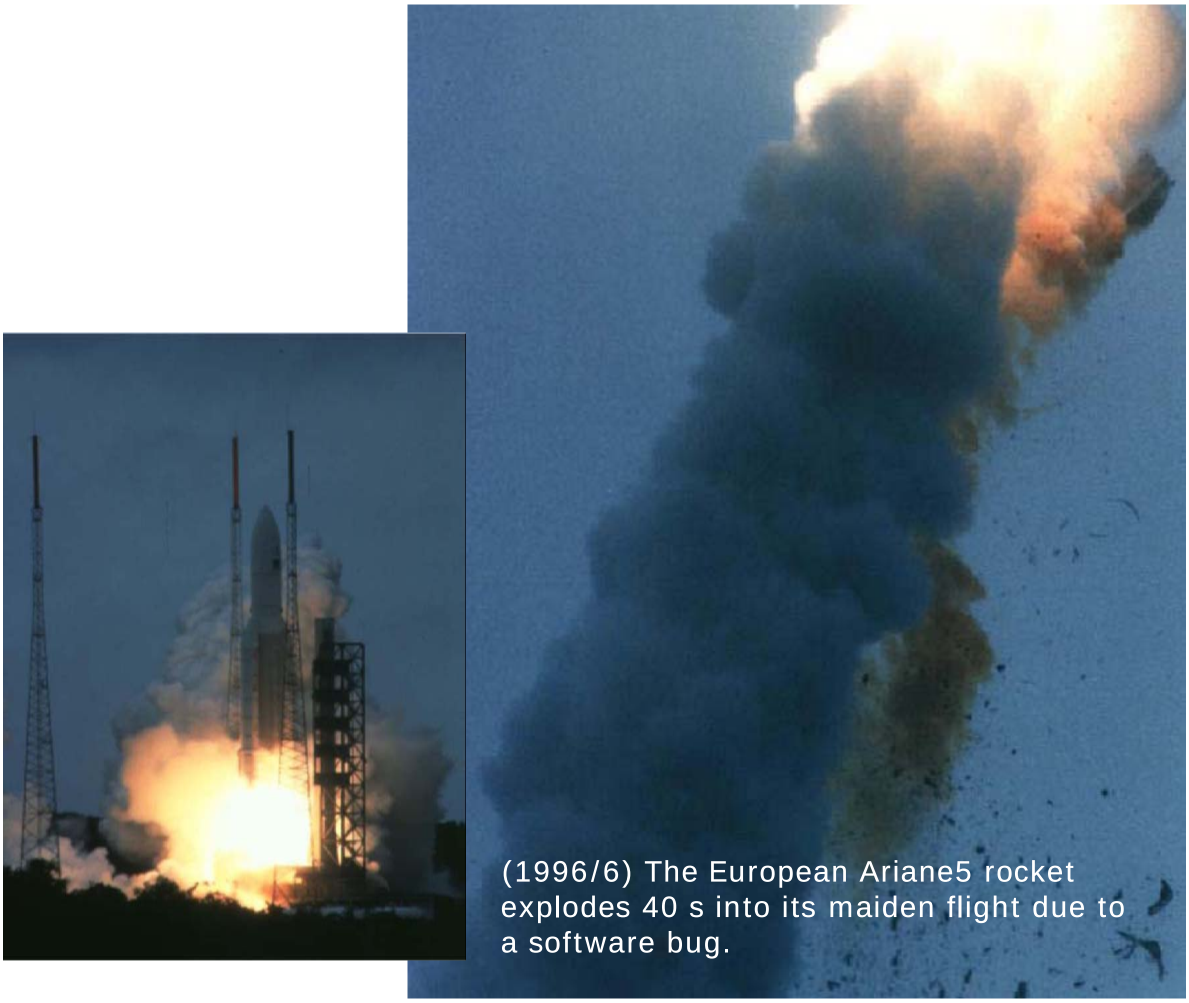
Formal Verification

- Course contents
  - Introduction
  - Boolean reasoning engines
  - Equivalence checking
  - Property checking
- Readings
  - Chapter 9

Outline

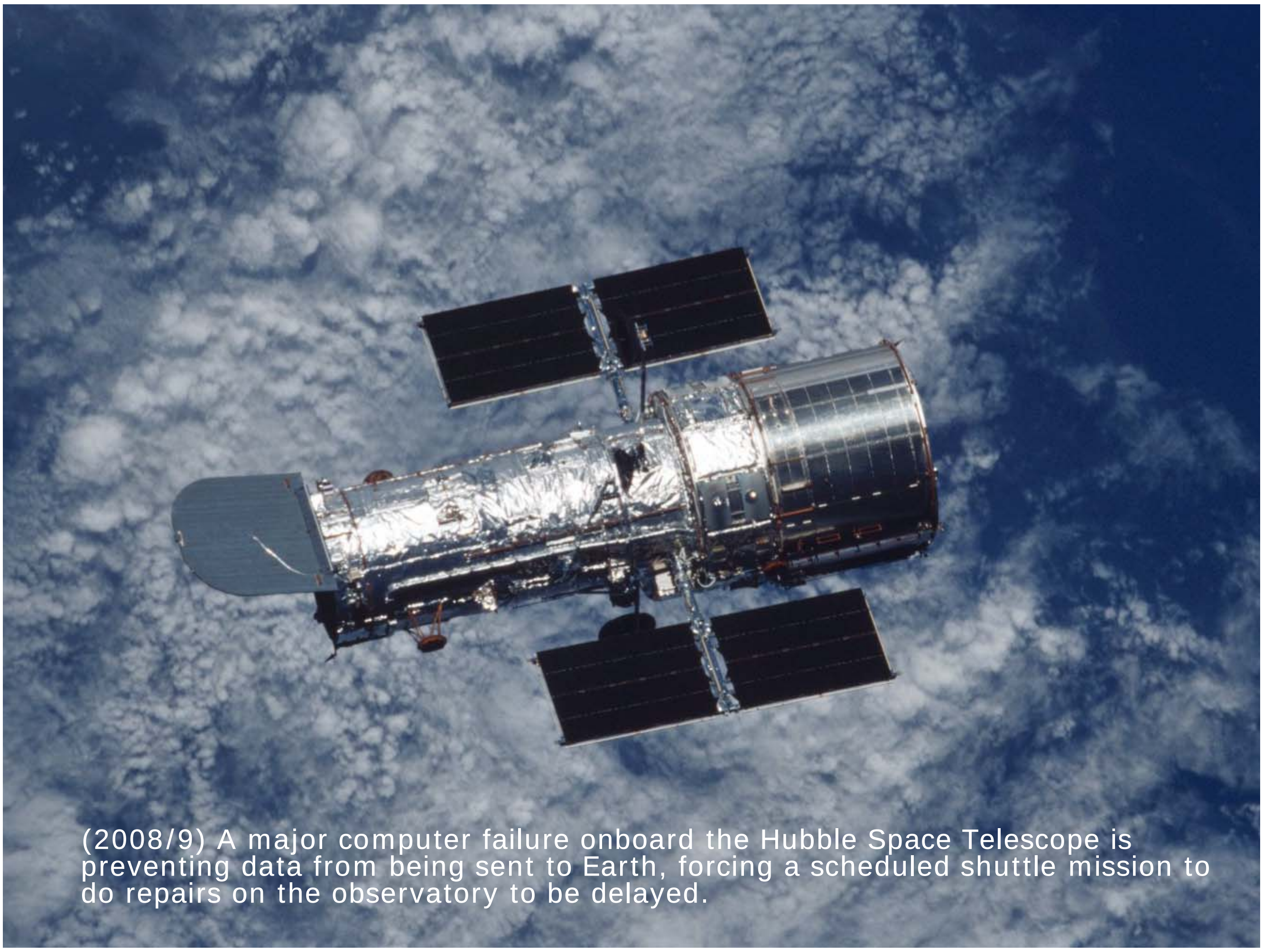
- Introduction
- Boolean reasoning engines
- Equivalence checking
- Property checking

(1995/1) Intel announces a pre-tax charge of 475 million dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors.



(1996/6) The European Ariane5 rocket explodes 40 s into its maiden flight due to a software bug.


003/45/7844
(2003/8) A programming error has been identified as the cause of the Northeast power blackout, which affected an estimated 10 million people in Canada and 45 million people in the U.S.
T GeoStar 45
15 EST 14 Aug. 2003

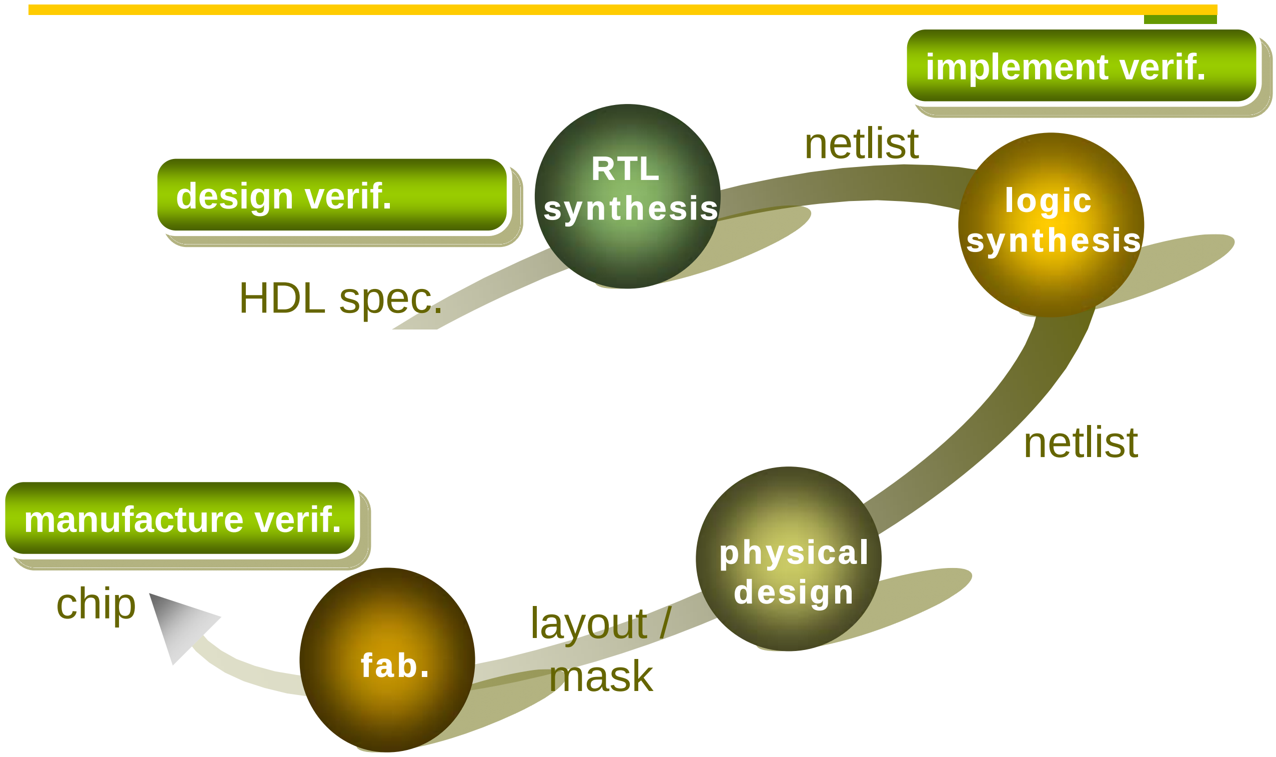

(2008/9) A major computer failure onboard the Hubble Space Telescope is preventing data from being sent to Earth, forcing a scheduled shuttle mission to do repairs on the observatory to be delayed.

Design vs. Verification

- Verification may take up to 70% of total development time of modern systems !
  - This ratio is ever increasing
  - Some industrial sources show 1:3 head-count ratio between design and verification engineers

- Verification plays a key role to reduce design time and increase productivity

IC Design Flow and Verification



Scope of Verification

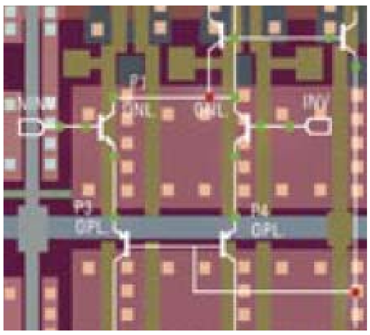
- Design flow
  - A series of transformations from abstract specification all the way to layout
- Verification enters design flow in almost all abstraction levels
  - Design verification
    - Functional property verification (main focus)
  - Implementation verification
    - Functional equivalence verification (main focus)
    - Physical verification
    - Timing verification
    - Power analysis
    - Signal integrity check
      - Electro-migration, IR-drop, ground bounce, cross-talk, etc.
  - Manufacture verification
    - Testing

Verification

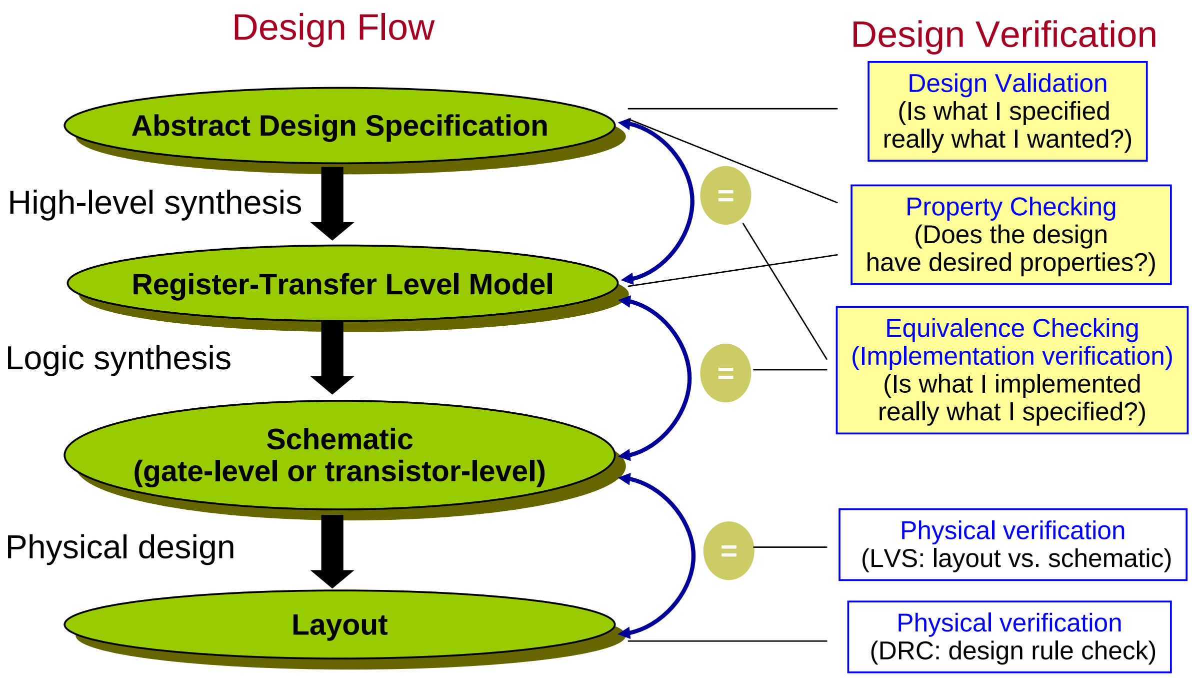
- Design/Implementation Verification

  Functional Verification
  - Property checking in system level
    - PSPACE-complete
  - Equivalence checking in RTL and gate level
    - PSPACE-complete

  Physical Verification
  - DRC (design rule check) and LVS (layout vs. schematic check) in layout level
    - Tractable
- Manufacture Verification
  - Testing
    - NP-complete
- "Verification" often refers to functional verification



Functional Verification



Functional Verification Approaches

- Simulation (software)
  - Incomplete (i.e., may fail to catch bugs)
  - Time-consuming, especially at lower abstraction levels such as gate- or transistor-level
  - Still the most popular way for design validation
- Emulation (hardware)
  - FPGA-based emulation systems, emulation system based on massively parallel machines (e.g., with 8 boards, 128 processors each), etc.
  - 2 to 3 orders of magnitude faster than software simulation
  - Costly and may not be easy-to-use
- Formal verification
  - a relatively new paradigm for property checking and equivalence checking
  - requires no input stimuli
  - perform exhaustive proof through rigorous logical reasoning

Informal vs. Formal Verification

- Informal verification
  - Functional simulation aiming at locating bugs
  - Incomplete
    - Show existence of bugs, but not absence of bugs
- Formal verification
  - Mathematical proof of design correctness
  - Complete
    - Show both existence and absence of bugs

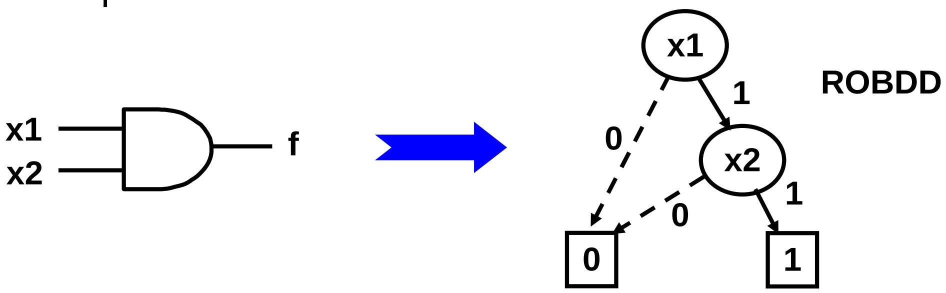
We will be focusing on formal verification

Outline

- Introduction
- Boolean reasoning engines
  - BDD
  - SAT
- Equivalence checking
- Property checking

Binary Decision Diagram (BDD)

- Basic features
  - ROBDD
    - Proposed by R.E. Bryant in 1986
    - A directed acyclic graph (DAG) representing a Boolean function f: $B^n \rightarrow B$
      - Each non-terminal node is a decision node associated with a input variable with two branches: 0-branch and 1-branch
      - Two terminal nodes: 0-terminal and 1-terminal
  - Example



Binary-Decision Diagram (BDD)

- Cofactor of Boolean function:
  - Positive cofactor w.r.t. x_i: $f_{xi} = f(x_1,\ldots, x_{i-1}, 1, x_{i+1},\ldots, x_n)$
  - Negative cofactor w.r.t. x_i: $f_{\neg xi} = f(x_1,\ldots, x_{i-1}, 0, x_{i+1},\ldots, x_n)$
  - Example

    $f = x_1' x_2' x_3' + x_1' x_2' x_3 + x_1 x_2' x_3 + x_1 x_2 x_3' + x_2 x_3$

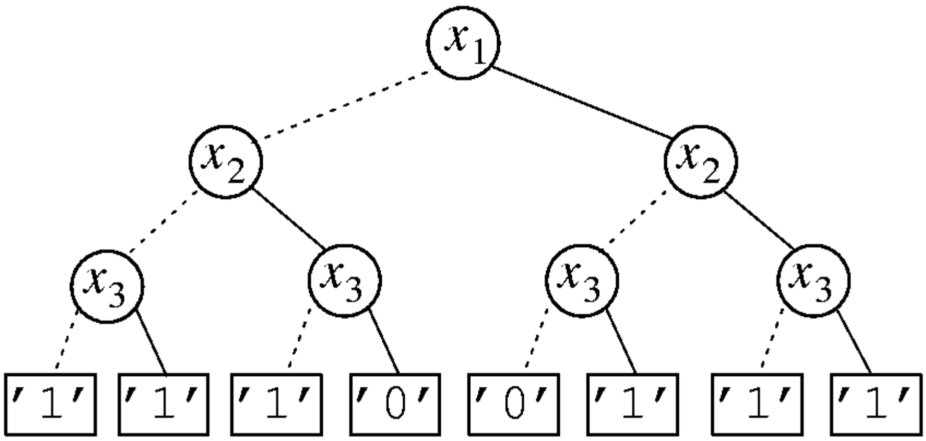
    $f_{x1} = x_2' x_3 + x_2 x_3' + x_2 x_3$

    $f_{x1'} = x_2' x_3' + x_2' x_3 + x_2 x_3$

- Shannon expansion: $f = x_i f_{xi} + x_i' f_{xi'}$
  - A complete expansion of a function can be obtained by successively applying Shannon expansion on all variables until either of the constant functions '0' or '1' is reached

Ordered BDD (OBDD)

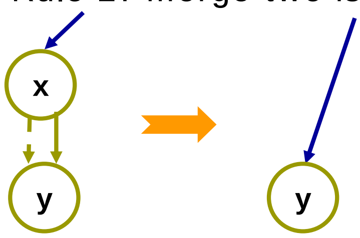
- Complete Shannon expansion can be visualized as a binary tree
  - Solid (dashed) lines correspond to the positive (negative) cofactor

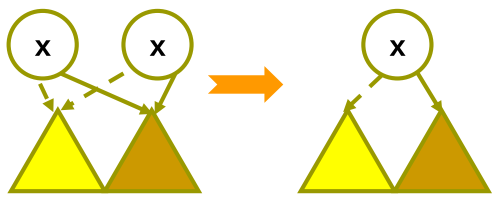


$$f = \overline{x}_1 \overline{x}_2 \overline{x}_3 + \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 \overline{x}_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3$$

Reduced OBDD (ROBDD)

- Reduction rules of ROBDD
  - Rule 1: eliminate a node with two identical children
  - Rule 2: merge two isomorphic sub-graphs





- Reduction procedure
  - Input: An OBDD
  - Output: An ROBDD
  - Traverse the graph from the terminal nodes towards to root node (i.e., in a bottom-up manner) and apply the above reduction rules whenever possible

ROBDD

- An OBDD is a directed tree $G(V,E)$
- Each vertex $v \in V$ is characterized by an associated variable $\phi(v)$, a *high* subtree $\eta(v)$ (high(v), the 1-branch) and a *low* subtree $\lambda(v)$ (low(v), the 0-branch)
- Procedure to reduce an OBDD:
  - Merge all identical leaf vertices and appropriately redirect their incoming edges
  - Proceed from bottom to top, process all vertices: if two vertices u and v are found for which $\phi(u)$ **=** $\phi(v)$, $\eta(u)$ **=** $\eta(v)$, and $\lambda(u) = \lambda(v)$, merge u and v and redirect incoming edges
  - For vertices v for which $\eta(v)$ **=** $\lambda(v)$, remove v and redirect its incoming edges to $\eta(v)$

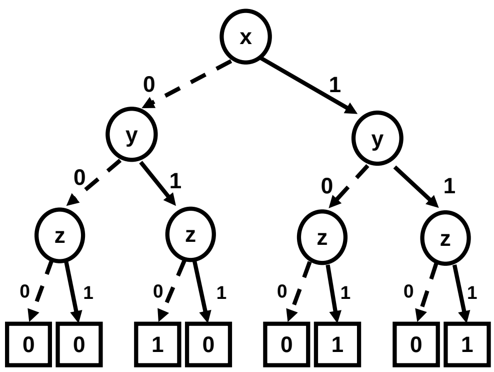
ROBDD

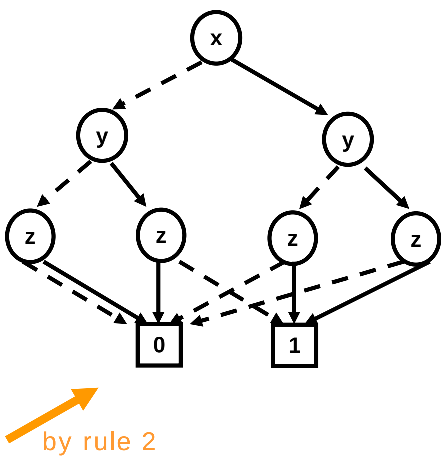
- Example
  - f = x'yz' + xz
  - variable order: x < y < z

Truth table

xyz	f
000	0
001	0
010	1
011	0
100	0
101	1
110	0
111	1

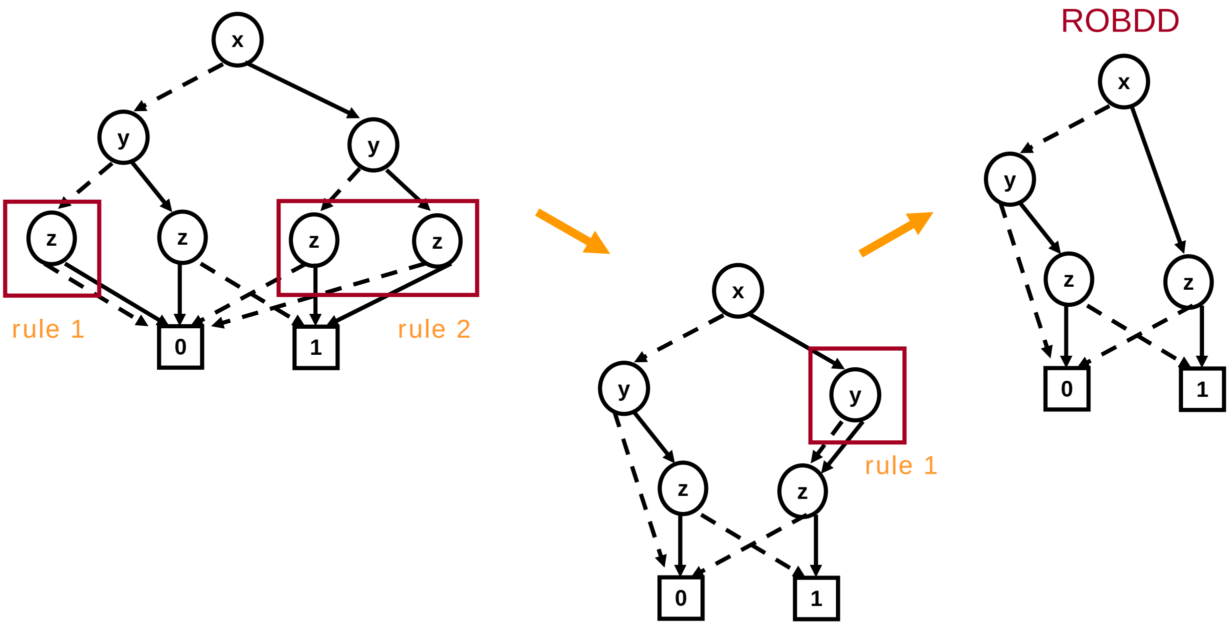
OBDD





ROBDD

Example (cont'd)



Canonicity

Canonicity requirements

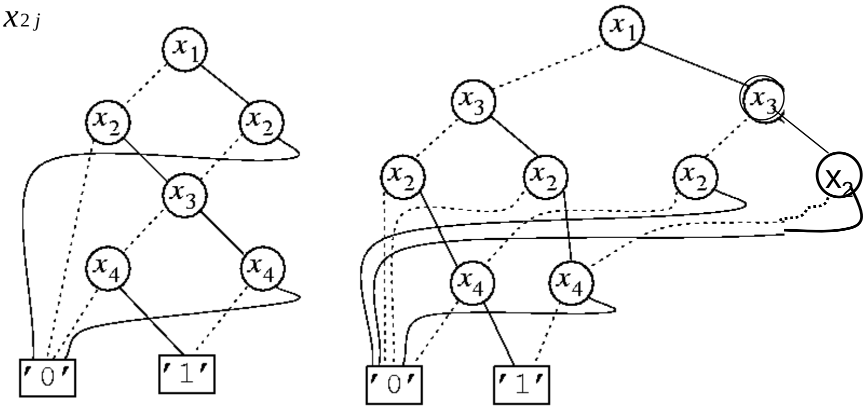
- A BDD representation is not canonical for a given Boolean function unless the following constraints are satisfied:

1. Simple BDD – each variable can appear only once along each path from the root to a leaf
2. Ordered BDD – Boolean variables are ordered in such a way that if the node labeled x_i has a child labeled x_k, then order(x_i) < order(x_k)
3. Reduced BDD – no two nodes represent the same function, i.e., redundancies are removed by sharing isomorphic sub-graphs

ROBDD Properties

- ROBDD is a canonical representation for a fixed variable ordering
- ROBDD is compact in representing many Boolean functions used in practice
- Variable ordering greatly affects the size of an ROBDD
  - E.g., the parity function of *k* bits:

$$f = \prod_{j=1}^{k} x_{2j-1} \oplus x_{2j}$$



Effects of Variable Ordering

- BDD size
  - Can vary from linear to exponential in the number of the variables, depending on the ordering
- Hard-to-build BDD
  - Datapath components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Heuristics of ordering
  - (1) Put the variable that influence most on top
  - (2) Minimize the distance between strongly related variables

    (e.g., x1x2 + x2x3 + x3x4)

    x1 < x2 < x3 < x4 is better than x1 < x4 < x2 < x3

BDD Package

- A BDD package refers to a software program that supports Boolean manipulation using ROBDDs. It has the following features:
  - It provides convenient API (application programming interface)
  - It supports the conversion between the external Boolean function representation and the internal ROBDD representation
  - Multiple Boolean functions are stored in shared ROBDD
  - It can create new functions from existing ones (e.g., $h = f \bullet g$)

BDD Data Structure

- A triplet (ϕ,η,λ) uniquely identifies an ROBDD vertex
- A unique table (implemented by a hash table) that stores all triplets already processed

```
struct vertex {
  char *φ;
  struct vertex *η, *λ;
   . . .
}
```

```
struct vertex *old_or_new(char *φ, struct vertex *η, *λ)
{
  if ("a vertex v = (φ, η, λ) exists")
    return v;
  else {
    v ← "new vertex pointing at (φ, η, λ)";
    return v;
  }
}
```

Building ROBDD

```
struct vertex *robdd_build(struct expr f, int i)
{
  struct vertex *η, *λ;
  struct char *φ;

  if (equal(f, '0'))
     return v0;
  else if (equal(f, '1'))
    return v1;
  else {
    φ ← π(i);
    η ← robdd_build(f_φ, i + 1);
    λ ← robdd_build(f_φ̄, i + 1);
    if (η = λ)
       return η;
    else
      return old_or_new(φ, η, λ);
  }
}
```

- The procedure directly builds the compact ROBDD structure
- A simple symbolic computation system is assumed for the derivation of the cofactors
- $\pi(i)$ gives the i^{th} variable from the top

Building ROBDD

- Example

$\text{robdd\_build}(\overline{x_1}\cdot\overline{x_3} + \overline{x_2}\cdot x_3 + x_1\cdot x_2, 1)$

$\xrightarrow{\eta}$ $\text{robdd\_build}(\overline{x_2}\cdot x_3 + x_2, 2)$

$\quad\xrightarrow{\eta}$ $\text{robdd\_build}('1', 3)$

$\quad\quad v_1$

$\quad\xrightarrow{\lambda}$ $\text{robdd\_build}(x_3, 3)$

$\quad\quad\xrightarrow{\eta}$ $\text{robdd\_build}('1', 4)$

$\quad\quad\quad v_1$

$\quad\quad\xrightarrow{\lambda}$ $\text{robdd\_build}('0', 4)$

$\quad\quad\quad v_0$

$\quad\quad v_2 = (x_3, v_1, v_0)$

$\quad v_3 = (x_2, v_1, v_2)$

$\xrightarrow{\lambda}$ $\text{robdd\_build}(\overline{x_3} + \overline{x_2}\cdot x_3, 2)$

$\quad\xrightarrow{\eta}$ $\text{robdd\_build}(\overline{x_3}, 3)$

$\quad\quad\xrightarrow{\eta}$ $\text{robdd\_build}('0', 4)$

$\quad\quad\quad v_0$

$\quad\quad\xrightarrow{\lambda}$ $\text{robdd\_build}('1', 4)$

$\quad\quad\quad v_1$

$\quad\quad v_4 = (x_3, v_0, v_1)$

$\quad\xrightarrow{\lambda}$ $\text{robdd\_build}(\overline{x_3} + x_3, 3)$

$\quad\quad\xrightarrow{\eta}$ $\text{robdd\_build}('1', 4)$

$\quad\quad\quad v_1$

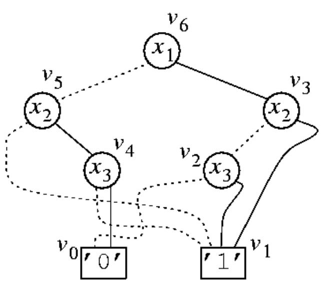
$\quad\quad\xrightarrow{\lambda}$ $\text{robdd\_build}('1', 4)$

$\quad\quad\quad v_1$

$\quad\quad v_1$

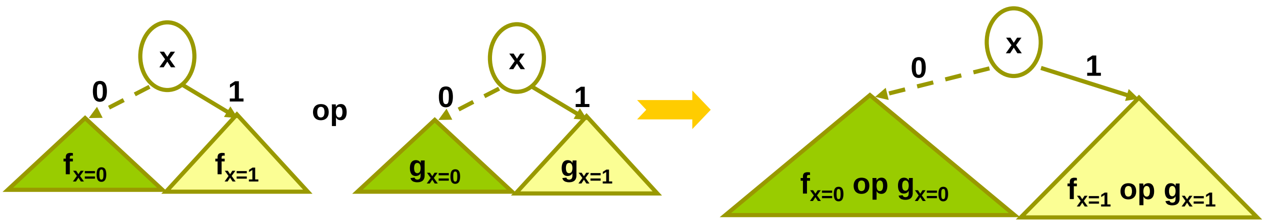
$\quad v_5 = (x_2, v_4, v_1)$

$v_6 = (x_1, v_3, v_5)$



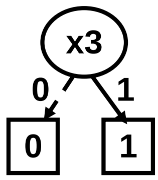
Recursive BDD Operation

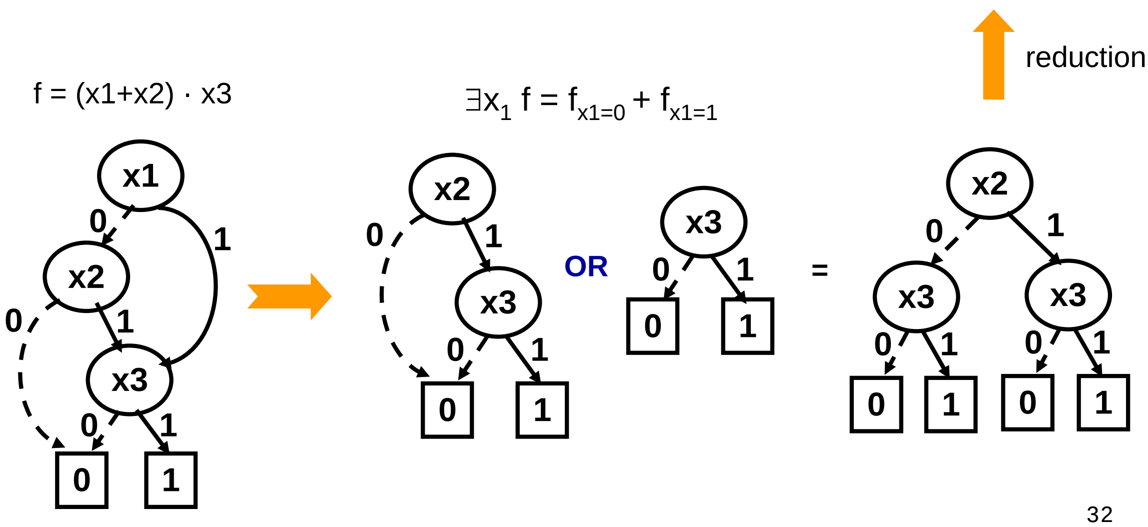
- Construct the ROBDD $h = f <op> g$ from two existing ROBDDs f and g, where <op> is a binary Boolean operator (e.g. AND, OR, NAND, NOR)
  - A recursive procedure on each variable x
    - $h \quad = x \cdot h_{x=1} + x' \cdot h_{x=0}$
      $= x \cdot (f <op> g)_{x=1} + x' \cdot (f <op> g)_{x=0}$
      $= x \cdot (f_{x=1} <op> g_{x=1}) + x' (f_{x=0} <op> g_{x=0})$
      - $(f <op> g)_x = (f_x <op> g_x)$ for <op> = AND, OR, NAND, NOR



Recursive BDD Operation

- Existential quantification
  Let $\exists x_1 [f(x_1,y_1, \ldots ,y_n)] = g(y_1, \ldots ,y_n)$.
  Then $g(y_1, \ldots ,y_n) = 1$ iff
  $f(0,y_1, \ldots ,y_n)=1$ or $f(1,y_1, \ldots ,y_n)=1$





ROBDD Manipulation

- Separate algorithms could be designed for each operator on ROBDDs, such as AND, NOR, etc. However, the universal if-then-else operator '*ite*' is sufficient.
  z **=** *ite*(*f*,*g*,*h*), *z* equals *g* when *f* is true and equals *h* otherwise:
  - Example:

$$z = ite(f, g, h) = f \cdot g + \bar{f} \cdot h$$

$$z = f \cdot g = ite(f, g, '0')$$

$$z = f + g = ite(f, '1', g)$$

- The *ite* operator is well-suited for a recursive algorithm based on ROBDDs ($\phi(v) = x$):

$$v = ite(F, G, H) = (x, ite(F_x, G_x, H_x), ite(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$$

ITE Operator

- ITE operator ite(f,g,h) = fg + f'h can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of **B**^{2}:

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	f g	ite(f, g, 0)
0010	f > g	f g′	ite(f, g′, 0)
0011	f	f	f
0100	f < g	f′g	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	f ⊕ g	ite(f, g′, g)
0111	OR(f, g)	f + g	ite(f, 1, g)
1000	NOR(f, g)	(f + g)′	ite(f, 0, g′)
1001	XNOR(f, g)	f ⊕ g′	ite(f, g, g′)
1010	NOT(g)	g′	ite(g, 0, 1)
1011	f ≥ g	f + g′	ite(f, 1, g′)
1100	NOT(f)	f′	ite(f, 0, 1)
1101	f ≤ g	f′ + g	ite(f, g, 1)
1110	NAND(f, g)	(f g)′	ite(f, g′, 1)
1111	1	1	1

Recursive Formulation of ITE

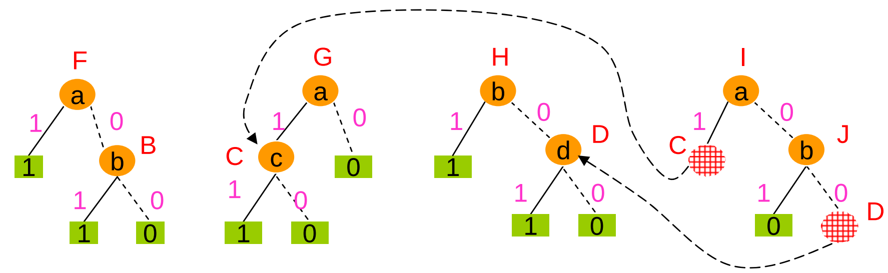
- Ite(f,g,h)

$$\begin{aligned}
&= f\,g + f'\,h \\
&= v\,(f\,g + f'\,h)_v + v'\,(f\,g + f'\,h)_{v'} \\
&= v\,(f_v\,g_v + f'_v\,h_v) + v'\,(f_{v'}\,g_{v'} + f'_{v'}\,h_{v'}) \\
&= \mathrm{ite}(v,\ \mathrm{ite}(f_v,g_v,h_v),\ \mathrm{ite}(f_{v'},g_{v'},h_{v'}))
\end{aligned}$$

where *v* is the top-most variable of BDDs *f, g, h*

ITE Operator

- Example



F,G,H,I,J,B,C,D
are pointers

$$\begin{aligned}
I &= ite\,(F, G, H) \\
&= ite\,(a, ite\,(F_a, G_a, H_a), ite\,(F_{\bar a}, G_{\bar a}, H_{\bar a})) \\
&= ite\,(a, ite\,(1, C, H), ite(B, 0, H)) \\
&= ite\,(a, C, ite\,(b, ite\,(B_b, 0_b, H_b), ite\,(B_{\bar b}, 0_{\bar b}, H_{\bar b})) \\
&= ite\,(a, C, ite(b, ite\,(1, 0, 1), ite\,(0, 0, D))) \\
&= ite\,(a, C, ite(b, 0, D)) \\
&= ite\,(a, C, J)
\end{aligned}$$

Check: F = a + b
G = ac
H = b + d
$ite(F, G, H) = (a + b)(ac) + a'b'(b + d) = ac + a'b'd$

ITE Operator

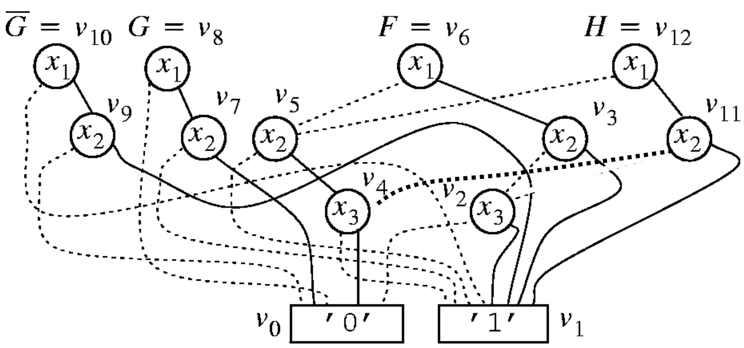
```
struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
  char x;
  struct vertex *η, *λ;

  if (F = v1)
    return G;
  else if (F = v0)
    return H;
  else if (G = v1 && H = v0)
    return F;
  else {
    x ← π(i);
    η ← apply_ite(Fx, Gx, Hx, i + 1);
    λ ← apply_ite(Fx̄, Gx̄, Hx̄, i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(x, η, λ);
  }
}
```

- ITE algorithm processes the variables in the order used in the BDD package
  - $\pi(i)$ gives the i^{th} variable from the top; $\pi^{-1}(x)$ gives the index position of variable x from the top
- Cofactor: Suppose F is the root vertex of the function for which F_x should be computed. Then
  $F_x = \eta(F)$ if $\pi^{-1}(\phi(F)) = i$
  - $F_{x'}$ can be calculated similarly
- The time complexity of the algorithm is $O(|F|\cdot|G|\cdot|H|)$

ITE Operator

- Example

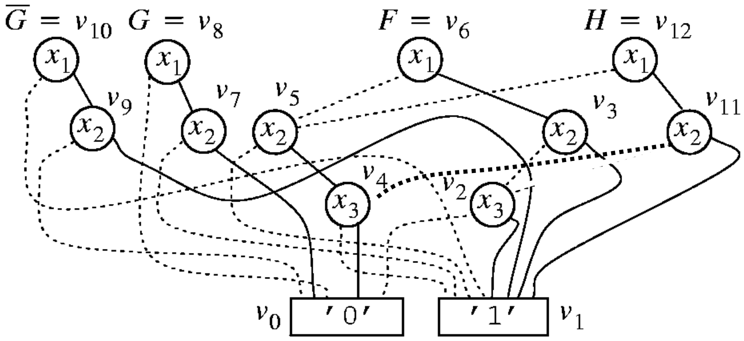


$\overline{G}$ = ite(G, 0, 1)

$\text{apply\_ite}(v_8, v_0, v_1, 1)$
$\xrightarrow{\eta} \text{apply\_ite}(v_7, v_0, v_1, 2)$
$\quad \xrightarrow{\eta} \text{apply\_ite}(v_0, v_0, v_1, 3)$
$\quad v_1$
$\quad \xrightarrow{\lambda} \text{apply\_ite}(v_1, v_0, v_0, 3)$
$\quad v_0$
$\quad v_9 = (x_2, v_1, v_0)$
$\xrightarrow{\lambda} \text{apply\_ite}(v_0, v_0, v_1, 2)$
$\quad v_1$
$v_{10} = (x_1, v_9, v_1)$

ITE Operator

- Example (cont'd)



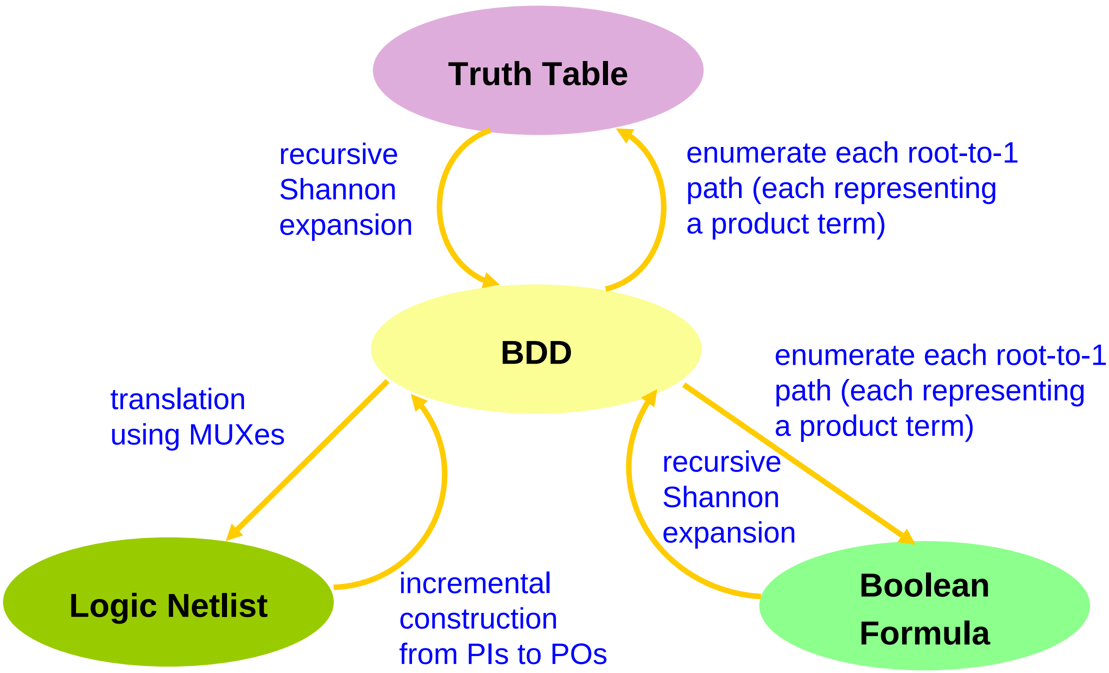
H = F ⊕ G
= ite(F, G, $\overline{G}$)

$\text{apply\_ite}(v_6, v_{10}, v_8, 1)$
$\xrightarrow{\eta} \text{apply\_ite}(v_3, v_9, v_7, 2)$
$\quad \xrightarrow{\eta} \text{apply\_ite}(v_1, v_1, v_0, 3)$
$\quad\quad v_1$
$\quad \xrightarrow{\lambda} \text{apply\_ite}(v_2, v_0, v_1, 3)$
$\quad\quad \xrightarrow{\eta} \text{apply\_ite}(v_1, v_0, v_1, 4)$
$\quad\quad\quad v_0$
$\quad\quad \xrightarrow{\lambda} \text{apply\_ite}(v_0, v_0, v_1, 4)$
$\quad\quad\quad v_1$
$\quad\quad v_4 = (x_3, v_0, v_1)$
$\quad v_{11} = (x_2, v_1, v_4)$
$\xrightarrow{\lambda} \text{apply\_ite}(v_5, v_1, v_0, 2)$
$\quad v_5$
$v_{12} = (x_1, v_{11}, v_5)$

BDD Memory Management

- Ordering
  - Finding the best ordering minimizing ROBDD sizes is intractable
  - Optimal ordering may change as ROBDDs are being manipulated
    - An ROBDD package may reorder the variables at different moments
    - It can move some variable closer to the top or bottom by remembering the best position, and repeat the procedure for other variables
- Garbage collection
  - Another important technique, in addition to variable ordering, for memory management

Data Type Conversion



Formula to BDD

Given a Boolean formula
$f = x3 \cdot (x1 + x2)$

↓

Use variable order: x1<x2<x3

↓

Shannon expansion on x1
$f = x1 \cdot f_{x1=1} + x1' \cdot f_{x1=0}$
$= x1 \cdot x3 + x1' \cdot x2 \cdot x3$

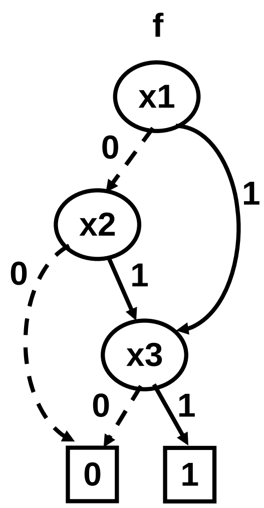
↓

Shannon expansion on x2 and x3
$f = x1 \cdot x3 + x1' \cdot (x2 \cdot x3 + x2' \cdot 0)$

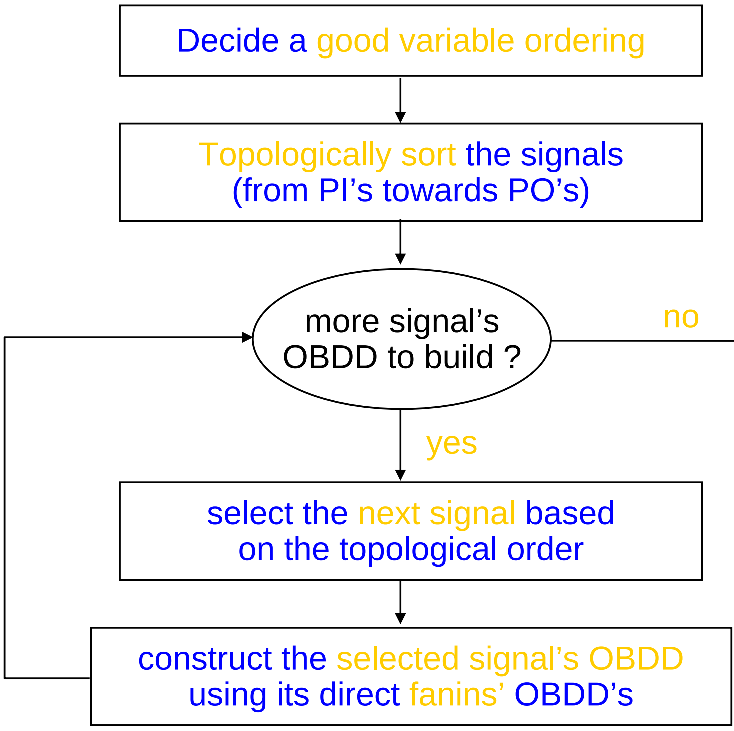
↓

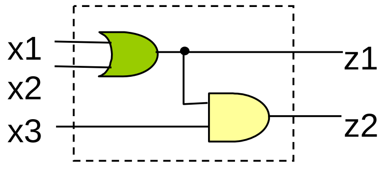
Perform reduction on the resulting BDD to a canonical form

a sequence of recursive Shannon expansions



Netlist to BDD





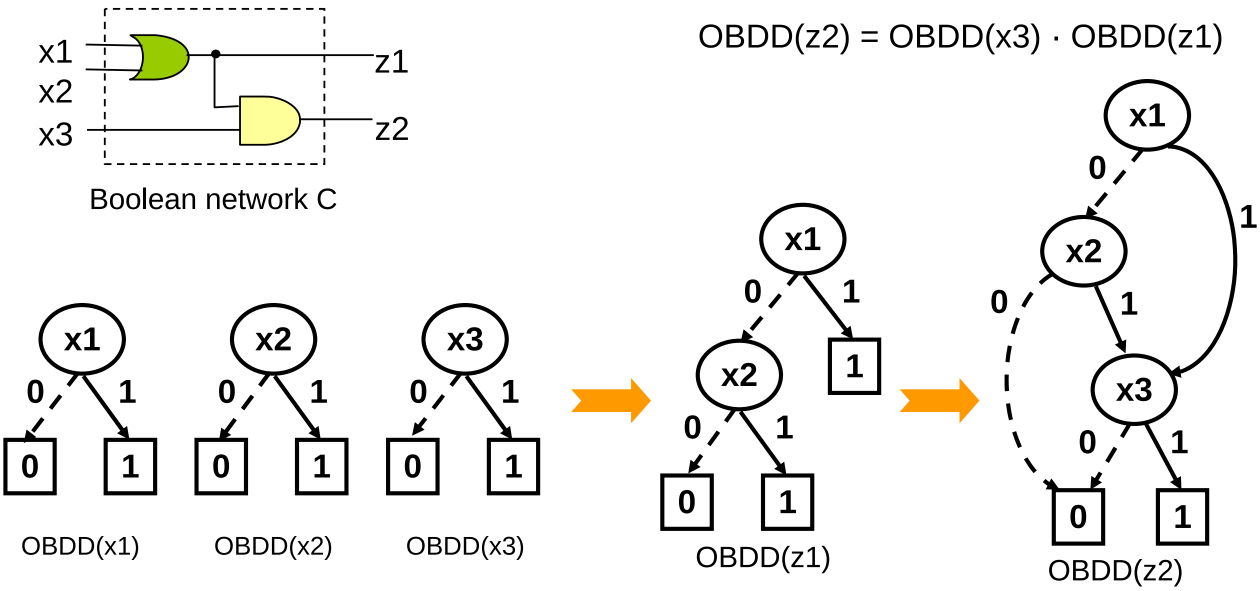
Boolean network C

Netlist to BDD

- Example

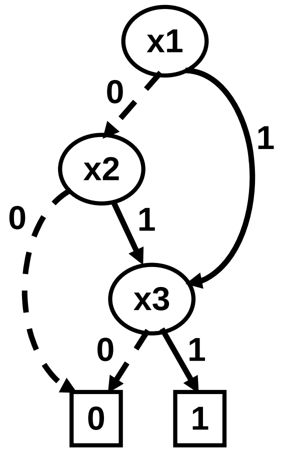
Topological order: {x1,x2,x3,z1,z2}
variable order: x1<x2<x3

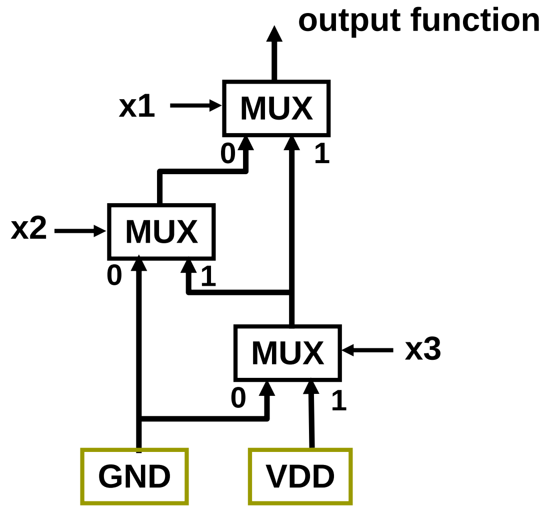
OBDD(z2) = OBDD(x3) · OBDD(z1)



BDD to Netlist

- MUX-based translation
  - replace each decision node by a MUX
  - replace 0-terminal by GND, and 1-terminal by VDD
  - reverse the direction of every edge
  - specify the root node as the output node





BDD Features

- Strengths
  - ROBDD is a compact representation for many Boolean functions
  - ROBDD is canonical, given a fixed variable ordering
  - Many Boolean operations are of polynomial time complexity in the input BDD sizes

- Weaknesses
  - In the worst case, the size of a BDD is $O(2^n)$ for n-input Boolean functions

BDD Applications

- Boolean function verification
  - Compare a specification f to an implementation g, assuming their ROBDDs are F and G, respectively.
    - For fully specified functions f and g, the verification is trivial (pointer comparison) because of the strong canonicity of the ROBDD
      - Strong canonicity: the representations of identical functions are the same
    - For an incompletely specified function $I = (f, d, \neg(f+d))$ with onset f, dc-set d, and offset $\neg(f+d)$. A completely specified function g correctly implements I if $(d + f \cdot g + \neg f \cdot \neg g)$ is a tautology, that is, $f \Rightarrow g \Rightarrow (f+d)$
- Satisfiability checking
  - A Boolean function f is satisfiable if there exists an input assignment for which f evaluates to '1'
  - Any Boolean function whose ROBDD is not equal to '0' is satisfiable

BDD Applications

- Min-cost satisfiability
  - Suppose that choosing a Boolean variable x_i to be '1' costs c_i. Then, the minimum-cost satisfiability problem asks to minimize: $\sum_i c_i \cdot u_i(x_i)$
    where $\mu(x_i) = 1$ when $x_i =$ '1' and $\mu(x_i) = 0$ when $x_i =$ '0'.
  - Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD with weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$, which can be solved in linear time
- Combinatorial optimization
  - Many combinatorial optimization problems can also be formulated in terms of the satisfiability problem
  - 0-1 integer linear programming can be formulated as a minimum-cost satisfiability problem although the translation may not be efficient
    - E.g., the constraint: $x_1 + x_2 + x_3 + x_4 = 3$ can be written as $(x_1+x_2)(x_1+x_3)(x_1+x_4)(x_2+x_3)(x_2+x_4)(x_3+x_4)(\neg x_1+\neg x_2+\neg x_3+\neg x_4)$

Outline

- Introduction
- Boolean reasoning engines
  - BDD
  - SAT
- Equivalence checking
- Property checking

SAT Solving

- SAT problem: Given a Boolean formula φ in CNF, find an input assignment such that φ valuates to true
- SAT solving is a decision procedure over CNFs

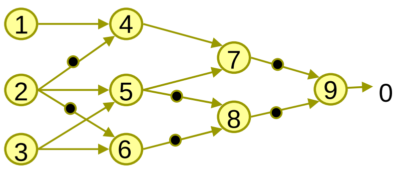
  Example

  $\varphi = (a+b'+c)(a'+b+c)(a+b'+c')(a+b+c)$

  is SAT (e.g. under a=1, b=1, c=0)
- SAT in CNF (POS) $\Leftrightarrow$ Tautology in DNF (SOP)
  - How about Tautology in CNF and SAT in DNF?

SAT Solving

- Given a circuit, suppose we would like to know if some signal is always zero. This can be formulated as a SAT problem if we can covert the circuit to an CNF.

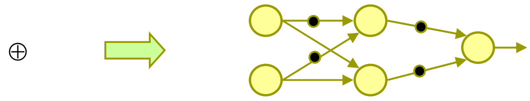


Is output always 0 ?

an AIG

Circuit to CNF

- Naive conversion of circuit to CNF:
  - Multiply out expressions of circuit until two level structure
  - Example: $y = x_1 \oplus x_2 \oplus x_2 \oplus \ldots \oplus x_n$ (Parity function)
    - circuit size is linear in the number of variables

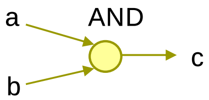


    - generated chess-board Karnaugh map
    - CNF (or DNF) formula has 2^{n-1} terms (exponential in #vars)

- Better approach:
  - Introduce one variable per circuit vertex
  - Formulate the circuit as a conjunction of constraints imposed on the vertex values by the gates
  - Uses more variables but size of formula is linear in the size of the circuit

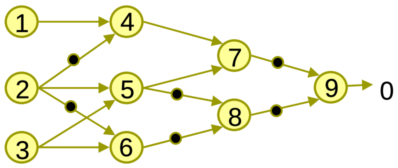
Circuit to CNF

- Example
  - Single gate:



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

  - Circuit of connected gates:



Is output always 0 ?

Justify to “1”

$$(\neg 1 + 2 + 4)(1 + \neg 4)(\neg 2 + \neg 4)$$
$$(\neg 2 + \neg 3 + 5)(2 + \neg 5)(3 + \neg 5)$$
$$(2 + \neg 3 + 6)(\neg 2 + \neg 6)(3 + \neg 6)$$
$$(\neg 4 + \neg 5 + 7)(4 + \neg 7)(5 + \neg 7)$$
$$(5 + 6 + 8)(\neg 5 + \neg 8)(\neg 6 + \neg 8)$$
$$(7 + 8 + 9)(\neg 7 + \neg 9)(\neg 8 + \neg 9)$$
$$(9)$$

Circuit to CNF

- Circuit to CNF conversion
  - can be done in linear size (with respect to the circuit size) if intermediate variables can be introduced
  - may grow exponentially in size if no intermediate variables are allowed

DPLL-Style SAT Solving

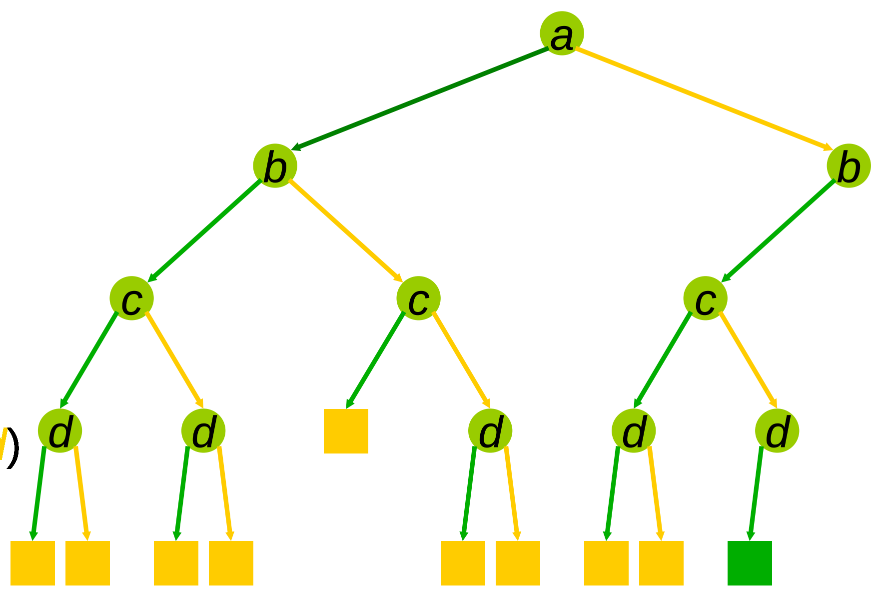
SAT(clause set S, literal v)

1. S := S_v //cofactor each clause of S w.r.t. v
2. If no clauses in S, return T
3. If a clause in S is empty (FALSE), return F
4. If S has a unit clause with literal u, then return SAT(S, u) //implication
5. Choose a variable x with value not yet assigned
6. If SAT(S, x), return T
7. If SAT(S, ¬x), return T
8. Return F

SAT Solving with Case Splitting

- Example

1 $(a + b + c)$
2 $(a + b + \neg c)$
3 $(\neg a + b + \neg c)$
4 $(a + c + d)$
5 $(\neg a + c + d)$
6 $(\neg a + c + \neg d)$
7 $(\neg b + \neg c + \neg d)$
8 $(\neg b + \neg c + d)$



Source: Karem A. Sakallah, Univ. of Michigan

SAT Solving with Implication

- Implication in a CNF formula are caused by unit clauses
  - A unit clause is a clause in which all literals except one are assigned (to be false)
    - The value of the unassigned variable is implied

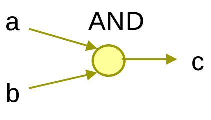
      Example

      $(a+\neg b+c)$

      $a=0,\ b=1 \Rightarrow c=1$

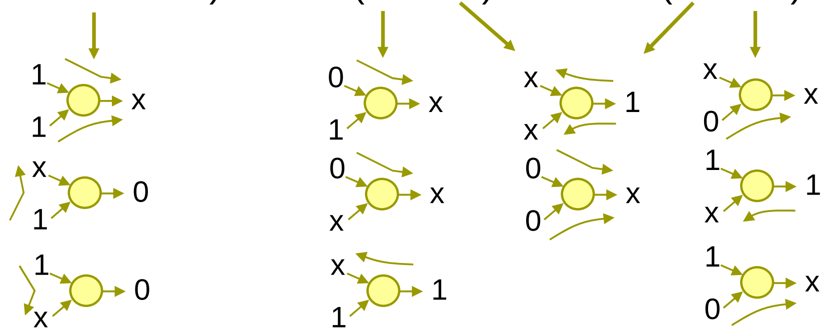
Implications in CNF

- Example



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

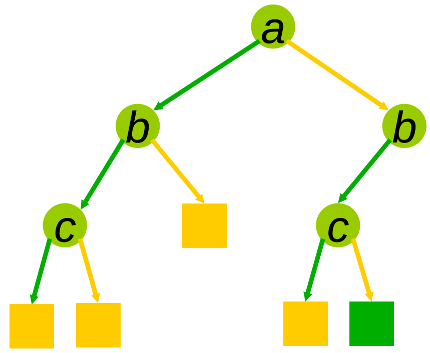
Implications: $(\neg a + \neg b + c)$ $(a + \neg c)$ $(b + \neg c)$

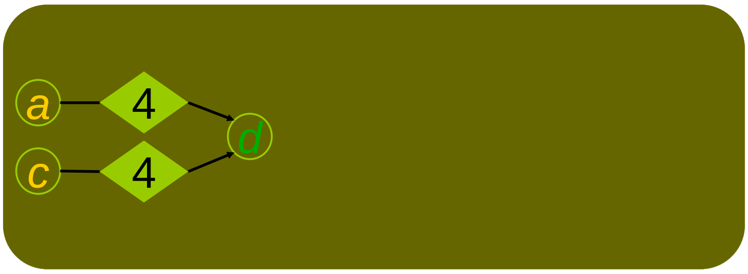


SAT Solving with Implication

- Example

1 $(a + b + c)$
2 $(a + b + \neg c)$
3 $(\neg a + b + \neg c)$
4 $(a + c + d)$
5 $(\neg a + c + d)$
6 $(\neg a + c + \neg d)$
7 $(\neg b + \neg c + \neg d)$
8 $(\neg b + \neg c + d)$



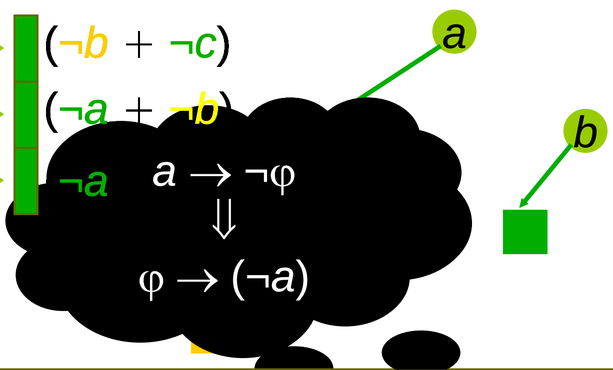


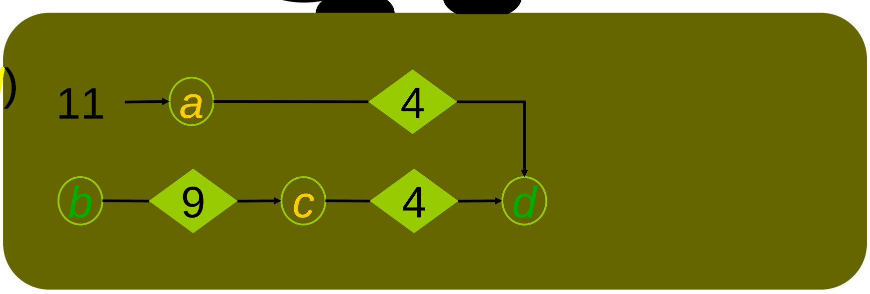
Source: Karem A. Sakallah, Univ. of Michigan

SAT Solving with Learning

- Example

1 $(a + b + c)$
2 $(a + b + \neg c)$
3 $(\neg a + b + \neg c)$
4 $(a + c + d)$
5 $(\neg a + c + d)$
6 $(\neg a + c + \neg d)$
7 $(\neg b + \neg c + \neg d)$
8 $(\neg b + \neg c + d)$
9 $(\neg b + \neg c)$
10 $(\neg a + \neg b)$
11 $\neg a$





Source: Karem A. Sakallah, Univ. of Michigan

Implementation Issues

- Track sensitivity of clauses for changes (two-literal-watch scheme)
  - clause with all literals but one assigned → implication
  - clause with all literals but two assigned → sensitive to a change of either literal
  - all other clauses are insensitive and need not be observed

- Learning:
  - learned implications are added to the CNF formula as additional clauses
    - limit the size of the clause
    - limit the "lifetime" of a learned clause, will be removed after some time

Quantification over CNF and DNF

- Recall a quantified Boolean formula (QBF) is
  $Q_1\, x_1,\ Q_2\, x_2,\ \ldots,\ Q_n\, x_n.\ \varphi$
  where Qi is either a existential ($\exists$) or universal quantifier ($\forall$), x_i is a Boolean variable, and φ is a Boolean formula.

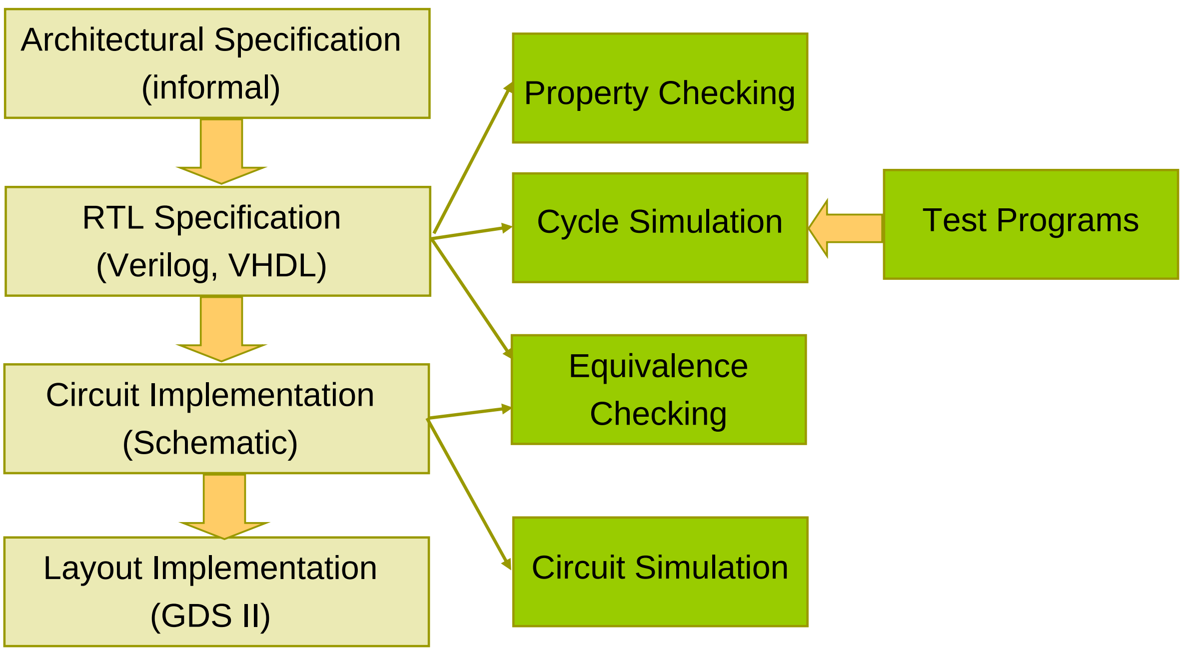
- Existential (respectively universal) quantification over DNF (respectively CNF) is easy
  - One approach to quantifier elimination is by back-and-forth CNF-DNF conversion!

- Solving QBFs with QBF-solvers

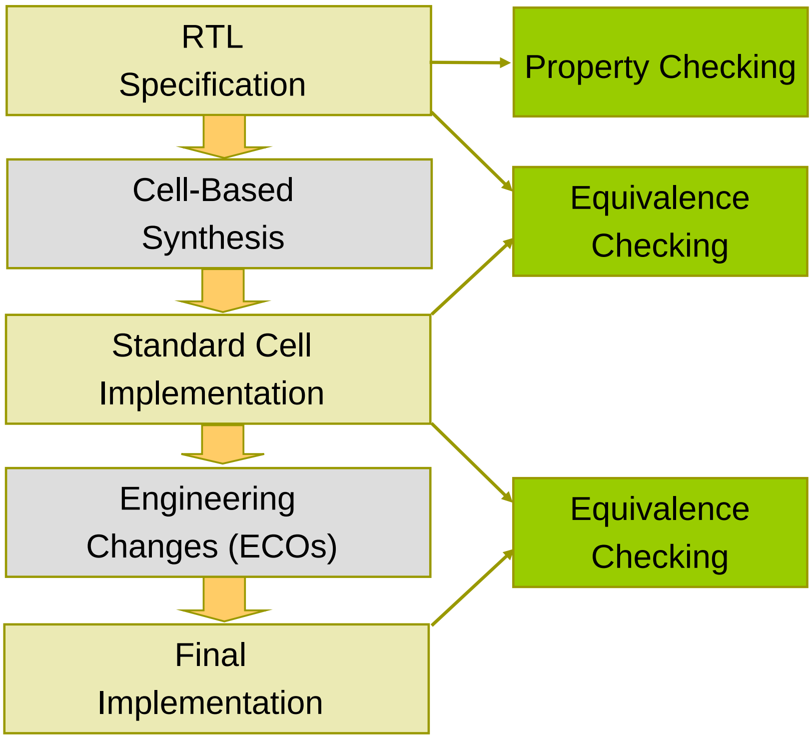
Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
- Property checking

Equivalence Checking in Microprocessor Design



Equivalence Checking in ASIC Design



Equivalence Checking

- Equivalence checking is one of the most important problem in design verification
  - It ensures logic transformation process (e.g. two-level, multi-level logic minimization, retiming and resynthesis, etc.) does not introduce errors

- Two types of equivalence checking
  - Combinational equivalence checking
    - Check if two combinational circuits are equivalent
  - Sequential equivalence checking
    - Check if two sequential circuits are equivalent

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

History of Equivalence Checking

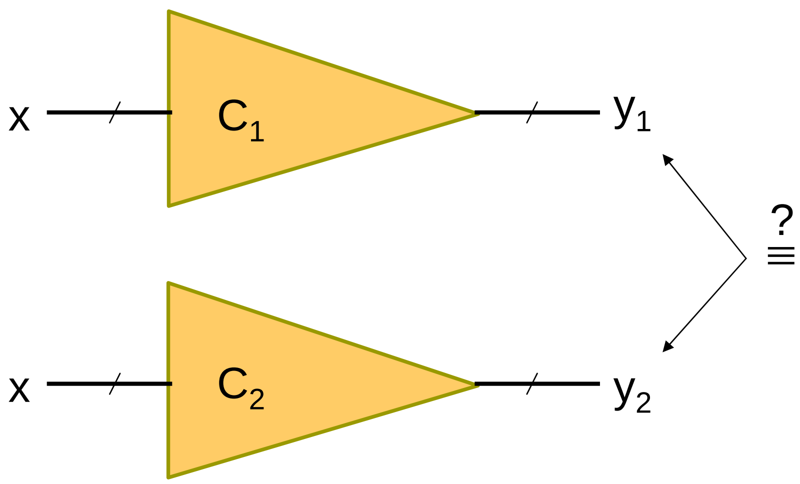
- SAS (IBM 1978 - 1994):
  - standard equivalence checking tool running on mainframes
  - based on the DBA algorithm ("BDDs in time")
  - verified manual cell-based designs against RTL spec
  - handling of entire processor designs
    - application of "proper cutpoints"
    - application of synthesis routines to make circuits structurally similar
    - special hacks for hard problems
- Verity (IBM 1992 - today):
  - originally developed for switch-level designs
  - today IBMs standard EC tool for any combination of switch-, gate-, and RTL designs

History of Equivalence Checking

- Chrysalis (1994 - Avanti - now Synopsys):
  - based on ATPG technology and cutpoint exploitation
  - very weak if many cutpoints present
  - did not adopt BDDs for a long time
- Formality (1997 - Synopsys)
  - multi-engine technology including strong structural matching techniques
- Verplex (1998 - now Cadence)
  - strong multi-engine based tool
  - heavy SAT-based
  - very fast front-end

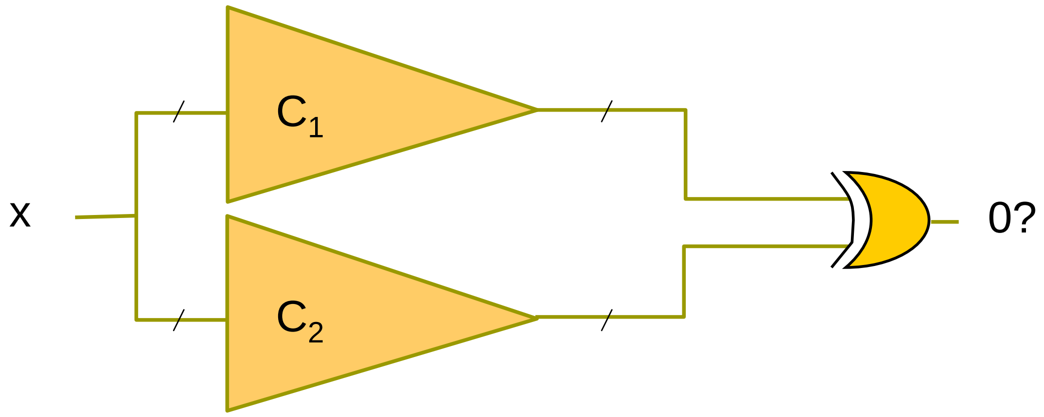
Combinational EC

- Given two combinational circuits C_1 and C_2, are their outputs equivalent under any possible input assignment?



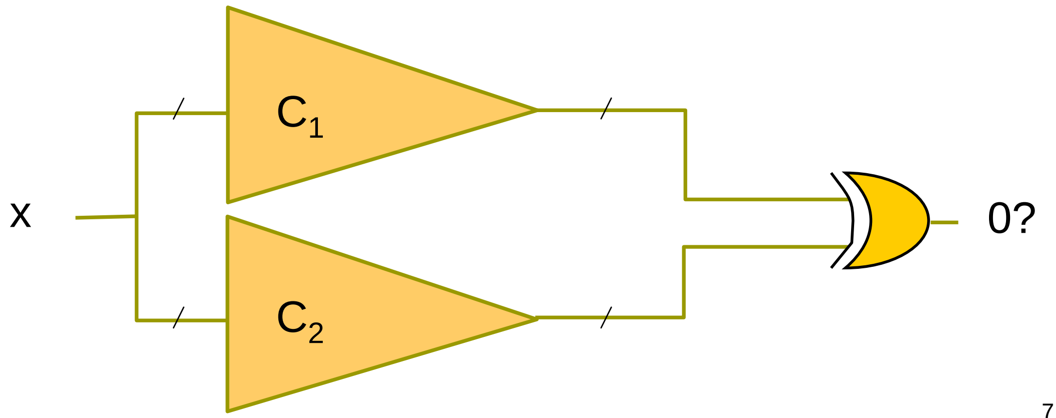
Miter for Combinational EC

- Two combinational circuits C_1 and C_2 are equivalent if and only if the output of their "miter" structure always produces constant 0



Approaches to Combinational EC

- Basic methods:
  - random simulation
    - good at identifying inequivalent signals
  - BDD-based methods
  - structural SAT-based methods



BDD-based Combinational EC

- Procedure
  1. Construct the ROBDDs F_1 and F_2 for circuits C_1 and C_2, respectively
     - Variable orderings of F_1 and F_2 should be the same
  2. Let $G = F_1 \oplus F_2$. If $G=0$, C_1 and C_2 are equivalent; otherwise, they are inequivalent
     - No false negative or false positive
       - False negative: circuits are equivalent; however, verifier fails to tell
       - False positive: circuits are inequivalent; however, verifier says otherwise

SAT-based Combinational EC

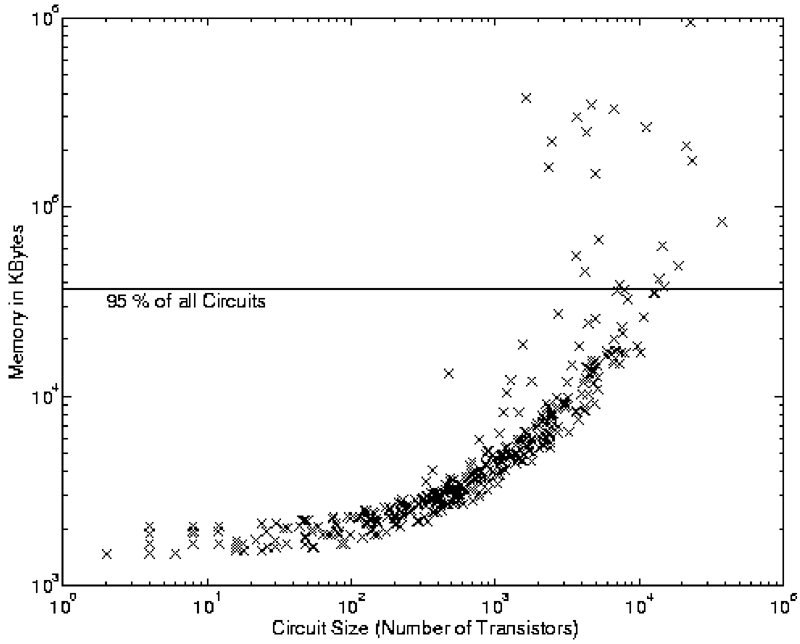
- Procedure
  1. Convert the miter structure into a CNF
  2. Perform SAT solving to verify if the output variable cannot be valuated to true under every input assignment (i.e. UNSAT)

Combinational EC

- Pure BDD and plain SAT solving cannot handle all logic cones
  - BDDs can be built for about 80% of the cones of high-speed designs and less for complex ASICs
  - plain SAT blows up in CPU time on a miter structure
- Contemporary method highly exploit structural similarities between two circuits to be compared

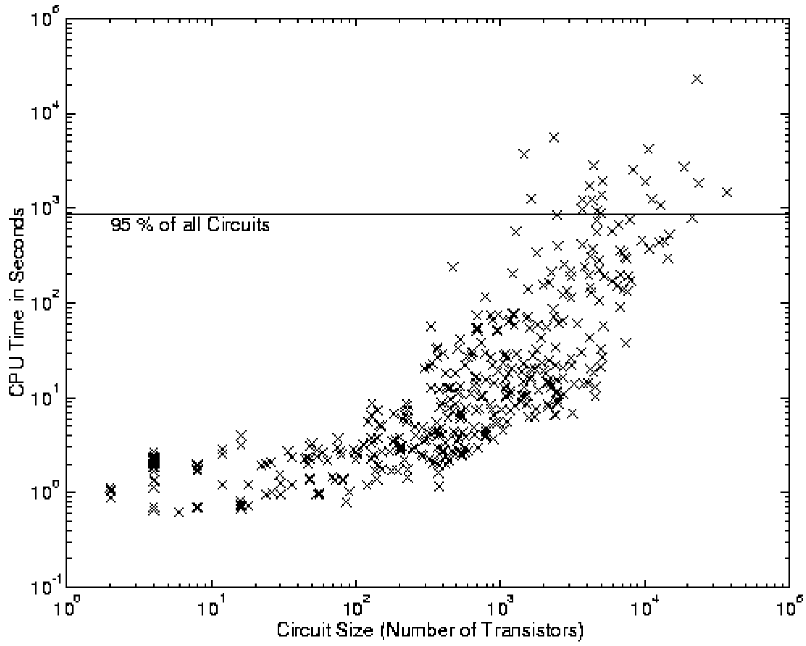
Combinational EC

- Memory statistics of BDD-based EC on a PowerPC processor design



Combinational EC

- Runtime statistics of BDD-based EC on a PowerPC processor design

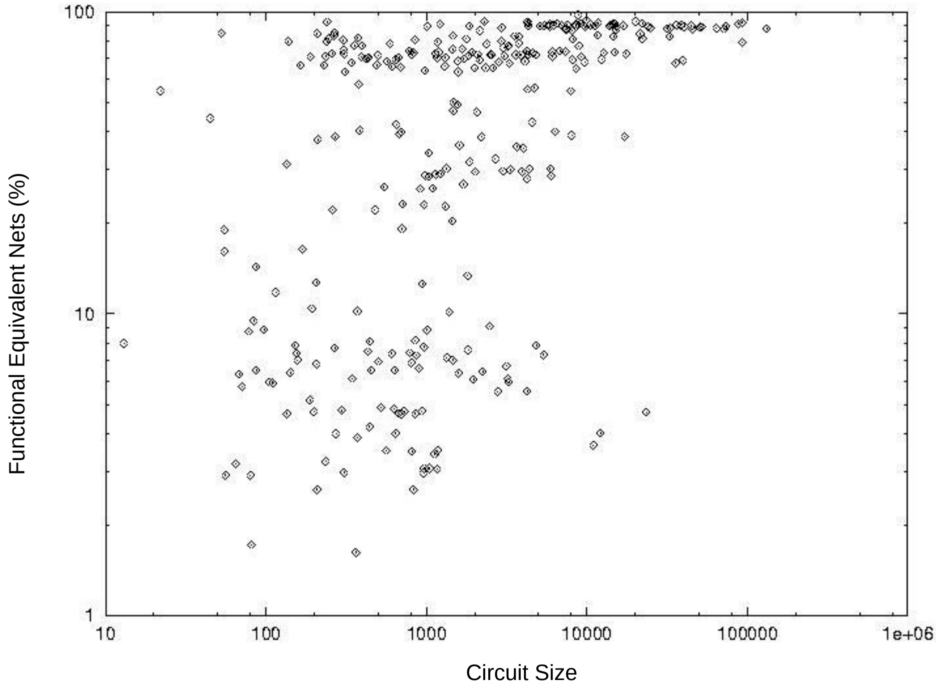


Necessity of Structure Similarity

- Pure BDDs are incapable of verifying equivalence of large circuits
  - Even more so for arithmetic circuits (e.g. BDDs blow up in representing multipliers)
- Identifying structure similarity helps simplify verification tasks
  - E.g. structure hashing in AIGs

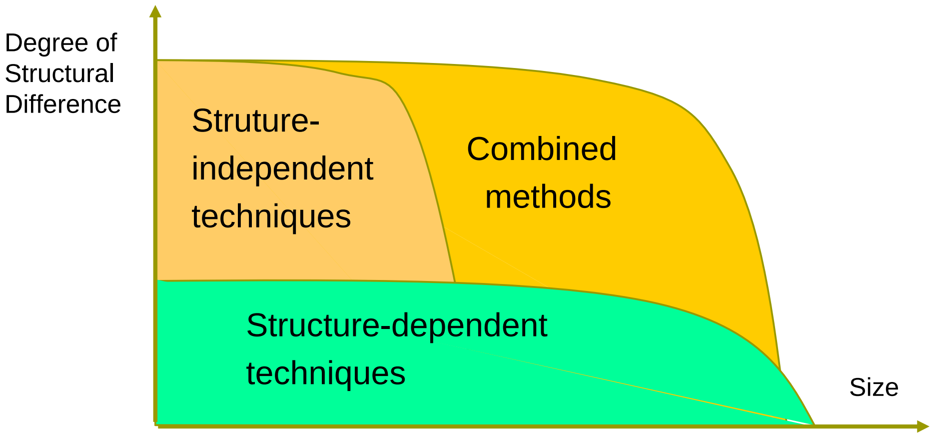
Combinational EC

- Evidence of vast existence of structure similarities



Structure and Verification

- Structure-independent techniques
  - Exhaustive simulation
  - Decision diagrams
- Structure-dependent techniques
  - Graph hashing
  - SAT based cutpoint identification



Summary

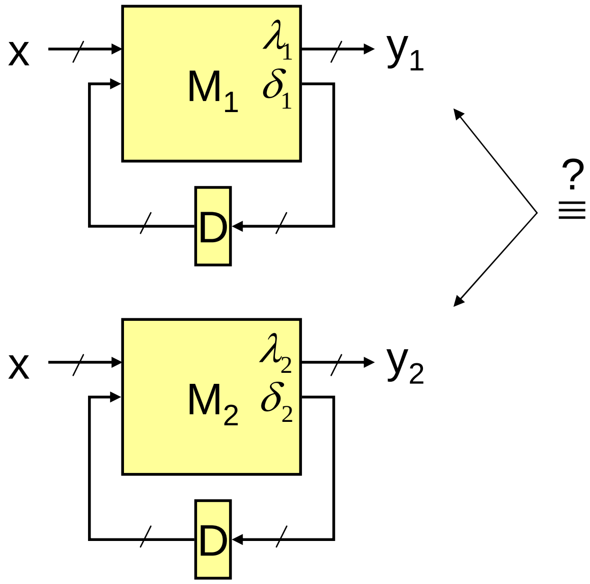
- Combinational EC is considered to be solvable in most industrial circuits (w/ multi-million gates)
  - Computational efforts scale almost linearly with the design size
  - Existence of structural similarities
    - Logic transformations preserve similarities to some extent
  - Hybrid engine of BDD, SAT, AIG, simulation, etc.
    - Cutpoint identification
- Unsolved for arithmetic circuits
  - Absence of structural similarities
    - Commutativity ruins internal similarities
  - Word- vs. bit-level verification

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

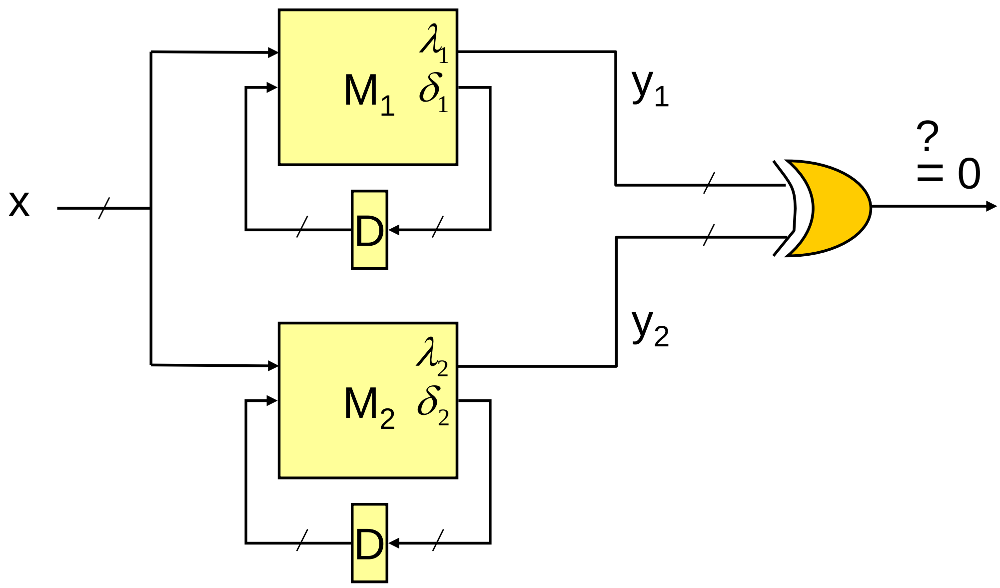
Sequential EC

- Given two sequential circuits (and thus FSMs), do they produce the same output sequence under any possible input sequence?



Miter for Sequential EC

- Two FSMs M_1 and M_2 are equivalent if and only if the output of their product machine always produces constant 0



Product Machine

- The product FSM $M_{1\times2}$ of FSMs $M_1 = (Q_1, I_1, \Sigma, \Omega, \delta_1, \lambda_1)$ and $M_2 = (Q_2, I_2, \Sigma, \Omega, \delta_2, \lambda_2)$ is a six-tuple $(Q_{1\times2}, I_{1\times2}, \Sigma, \Omega, \delta_{1\times2}, \lambda_{1\times2})$, where
  - State space $Q_{1\times2} = Q_1 \times Q_2$
  - Initial state set $I_{1\times2} = I_1 \times I_2$
  - Input alphabet Σ
  - Output alphabet $\{0,1\}$
  - Transition function $\delta_{1\times2} = (\delta_1, \delta_2)$
  - Output function $\lambda_{1\times2} = (\lambda_1 \oplus \lambda_2)$

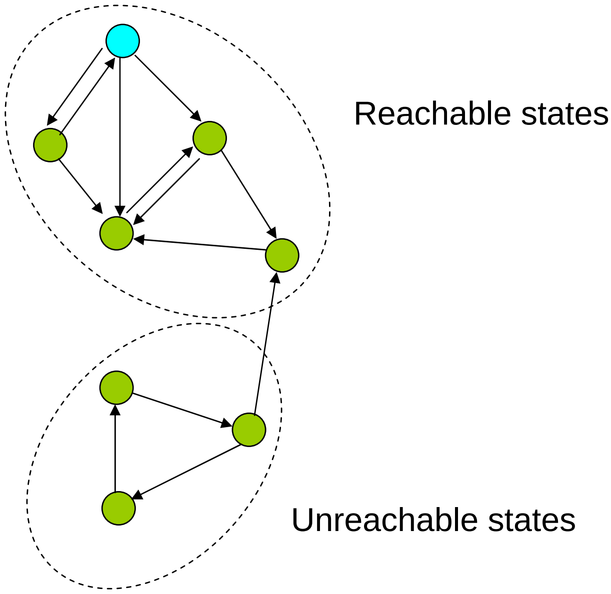
Sequential EC

- Approaches for combinational EC do not work for sequential EC because two equivalent FSMs need not have the same transition and output functions
  - False negatives may result from applying combinational EC on sequential circuits

- One solution to sequential EC is by reachability analysis
  - Two FSMs M_1 and M_2 are equivalent if and only if the output of their product FSM $M_{1\times2}$ is constant 0 under all input assignments and all reachable states of $M_{1\times2}$
  - Need to know the set of reachable states of $M_{1\times2}$

Reachability Analysis

- Given an FSM M = (Q, I, Σ, Ω, δ, λ) , which states are reachable from the initial state set I ?

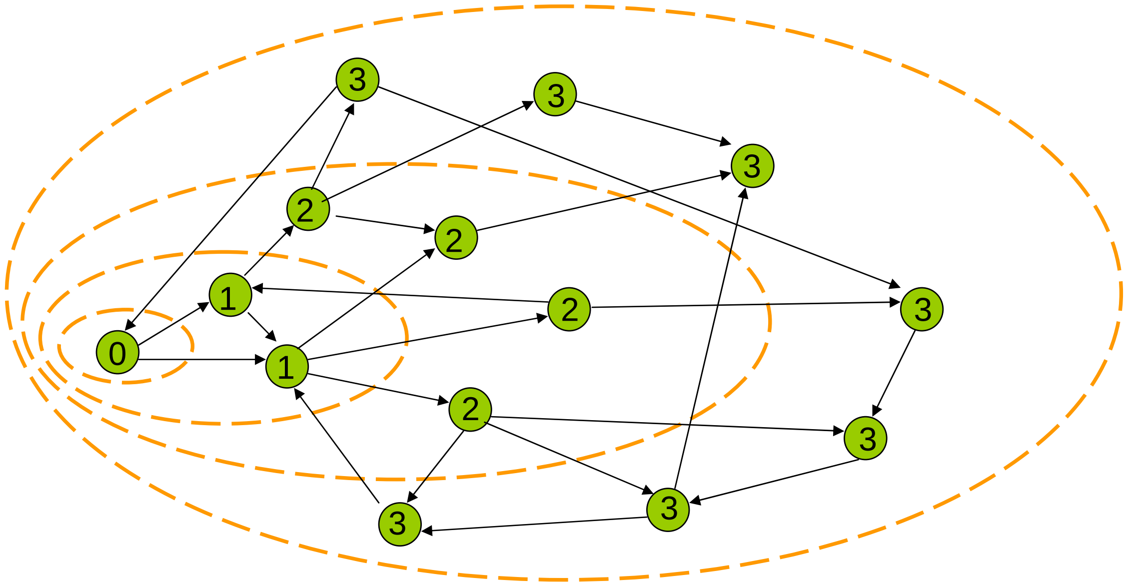


Symbolic Reachability Analysis

- Reachability analysis can be performed either explicitly (over a state transition graph) or implicitly (over transition functions or a transition relation)
  - Implicit reachability analysis is also called symbolic reachability analysis (often using BDDs and more recently SAT)

- Image computation is the core computation in symbolic reachability analysis

Reachability Onion Ring

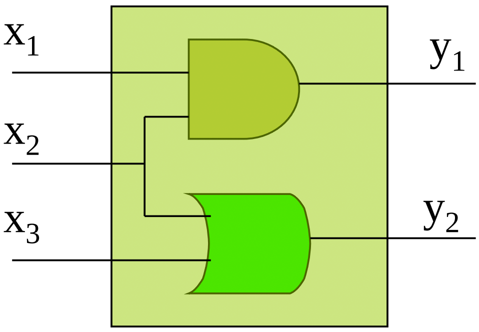


Computing Reachable States

- Input: Sequential system represented by a transition relation and an initial state (or a set of initial states)
  - Transition functions can be converted into a transition relation
- Computation: Image computation using Boolean operations on characteristic functions (representing state sets)
- Output: A characteristic function representing the set of reachable states

Relation

- **Definition**. **Relation** $R \subseteq X \times Y$ is a subset of the Cartesian product of two sets X and Y. If $(x,y) \in R$, then we alternatively write “x R y” meaning x is related to y by R.

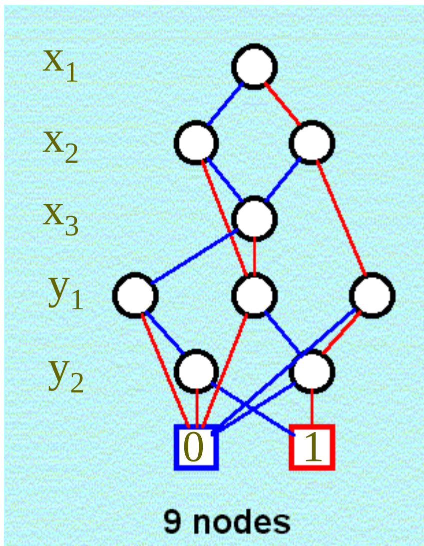


x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Characteristic Function

- Relation $R \subseteq X \times Y$ can be represented by a **characteristic function**: a Boolean function $F_R(x,y)$ taking value 1 for those $(x,y) \in R$ and 0 otherwise.

x_1	x_2	x_3	y_1	y_2	F
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1
other					0



Transition Relation

- Definition. A transition relation T of an FSM M = (Q, I, Σ, Ω, δ, λ) is a relation T $\subseteq$ (Σ x Q) x Q such that T(σ, q_1, q_2) = 1 iff there is a transition from q_1 to q_2 under input σ.
  - δ: (Σ x Q) → Q
  - T: (Σ x Q) x Q → {0,1}

  Assume δ = (δ_1, ..., δ_κ). Then

$$T(\vec{x},\vec{s},\vec{s}\,') = (s_1' \equiv \delta_1(\vec{x},\vec{s})) \wedge (s_2' \equiv \delta_2(\vec{x},\vec{s})) \wedge \cdots \wedge (s_k' \equiv \delta_k(\vec{x},\vec{s}))$$
$$= \prod_i (s_i' \equiv \delta_i(\vec{x},\vec{s}))$$

  where x, s, s' are primary-input, current-state, and next-state variables, respectively.

Quantified Transition Relation

- Definition

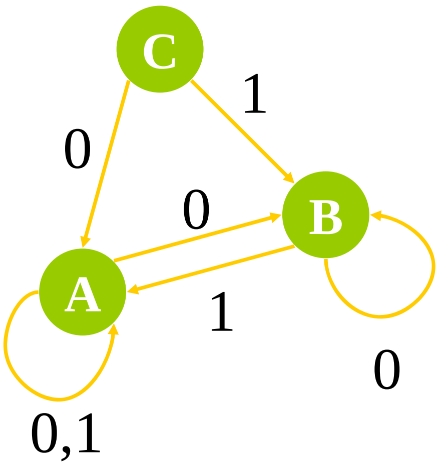
  Let M = (Q, I, Σ, Ω, δ, λ) be an FSM
  - Quantified transition relation $T_\exists$

$$T_\exists(\vec{s},\vec{s}\,') = \exists\vec{x}.(s_1' \equiv \delta_1(\vec{x},\vec{s})) \wedge (s_2' \equiv \delta_2(\vec{x},\vec{s})) \wedge \cdots \wedge (s_k' \equiv \delta_k(\vec{x},\vec{s}))$$
$$= \exists\vec{x}.\prod_i (s_i' \equiv \delta_i(\vec{x},\vec{s}))$$

    - (p,q) $\in T_\exists$ if there exists an input assignment bringing M from state p to state q
    - only concerns about the reachability of the FSM's transition graph

Transition Relation

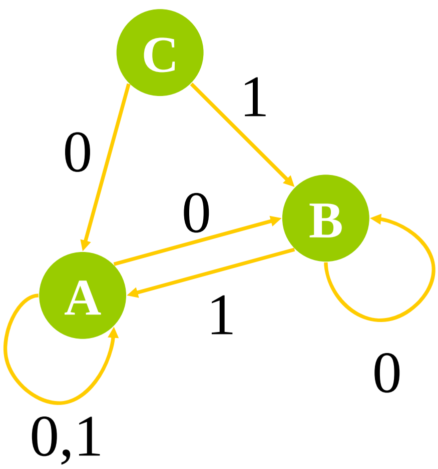
- Example



x	CS	$s_1\ s_2$	NS	$s_1'\ s_2'$	T
0	A	00	B	10	1
0,1	A	00	A	00	1
0	B	10	B	10	1
1	B	10	A	00	1
0	C	01	B	10	1
1	C	01	A	00	1
other					0

Transition Relation

- Example



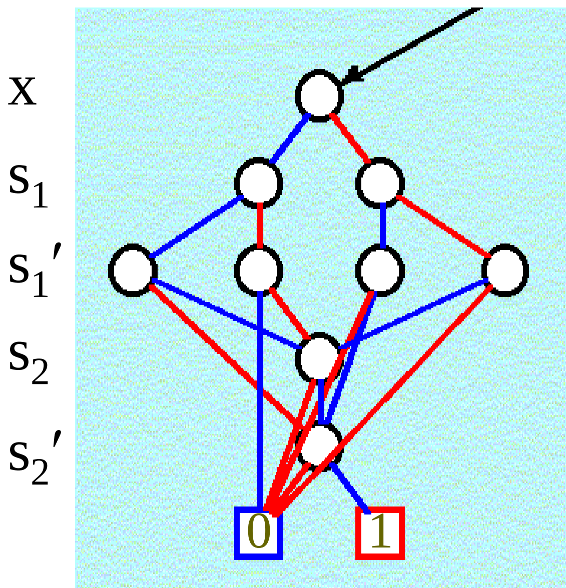


Image Computation

- Given a mapping of one Boolean space (input space) into another Boolean space (output space)
  - For a set of minterms (care set) in the input space
    - The image is the set of related minterms from the output space
  - For a set of minterms in the output space
    - The pre-image is the set of related minterms in the input space

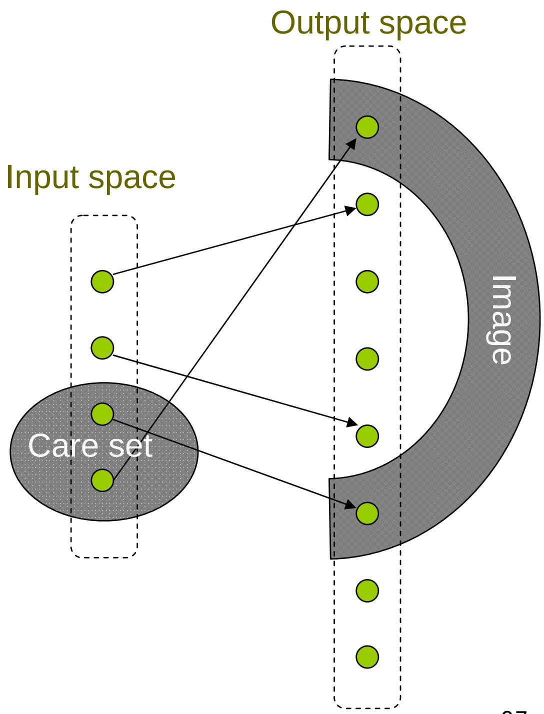
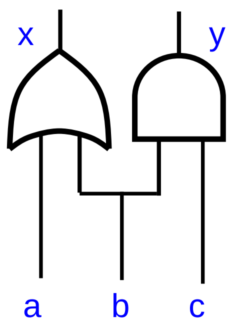


Image Computation

- Example



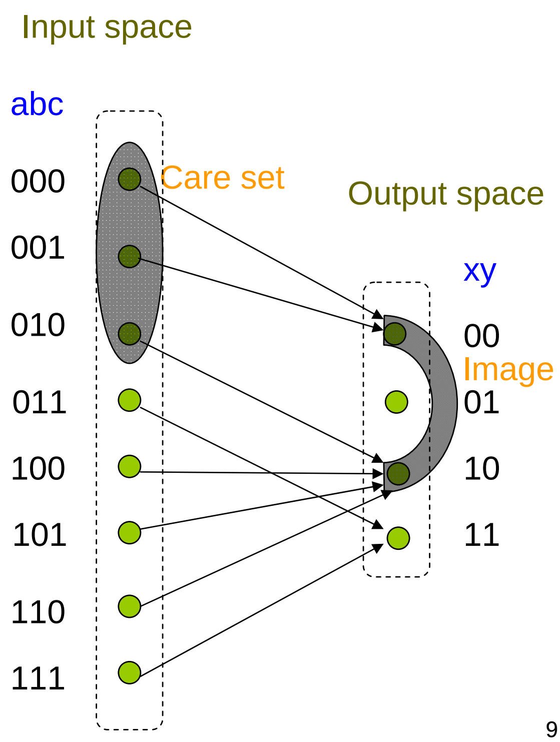


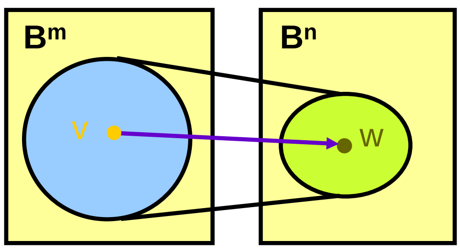
Image Computation

- $Image(C(x),T(x,y)) = \exists x\ [C(x) \wedge T(x,y)\ ]$
- Implicit methods by far outperform explicit ones
  - Successfully computing images with more than 2^{100} minterms in the input/output spaces
- Operations $\wedge$ and $\exists$ are basic Boolean manipulations and are implemented in BDD packages
  - To avoid large intermediate results (during and after the product computation), BDD AND-EXIST operation performs product and quantification in one pass over the BDD

Symbolic Image Computation

- Definition. Let F: $B^m \times B^n$ be a projection and C be a set of minterms in B^m. Then the image of C is the set
  $Img(C, F) = \{\ w \in B^n \mid (v, w) \in F \text{ and } v \in C\}$ in B^n.
- Characteristic function
  - for reachable next-state computation

$$N_i(\vec{s}\,') = Img(R_i(\vec{s}), T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}.(R_i(\vec{s}) \wedge T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}.(R_i(\vec{s}) \wedge (\exists \vec{x}.\prod_i (s_i' \equiv \delta_i(\vec{x}, \vec{s}))))$$

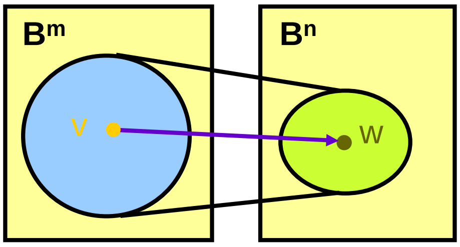


Symbolic Pre-Image Computation

- **Definition**. Let F: $B^m \times B^n$ be a projection and C be a set of minterms in B^m. Then the pre-image of C is the set PreImg(C, F) = { v ∈ B^m | (v, w) ∈ F and w ∈ C} in B^n.

- Characteristic Function
  - for reachable previous-state computation

$$N_i(\vec{s}) = PreImg(R_i(\vec{s}\,'), T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}\,'.(R_i(\vec{s}\,') \wedge T_\exists(\vec{s}, \vec{s}\,'))$$
$$= \exists \vec{s}\,'.(R_i(\vec{s}\,') \wedge (\exists \vec{x}.\prod_i (s_i' \equiv \delta_i(\vec{x}, \vec{s}))))$$



Reachability Analysis

```
ForwardReachability( Transition Relation T, Initial State I )
{
    i := 0
    R^i := I
    repeat
        R_new = Image( R^i, T );
        i := i + 1
        R^i := R^{i-1} ∨ R_new
    until R^i = R^{i-1}
    return R^i
}
```

- The procedures can be realized using BDD package.

- Backward reachability analysis can be done in a similar manner with pre-image computation and starting from final states to see if they can be reached from initial states.

Sequential Equivalence Checking

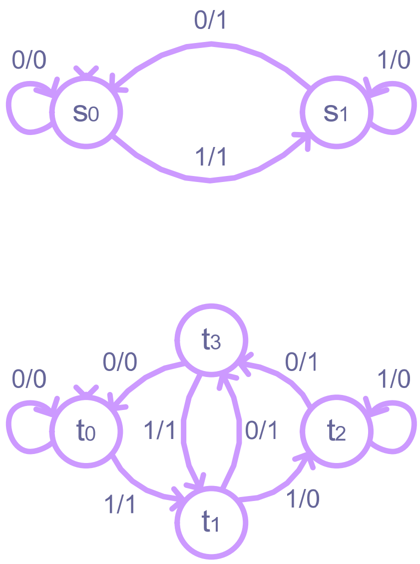
- Let R(s) be the characteristic function of the reachable state set of the product FSM $M_{1\times 2}$ obtained from forward reachability analysis. Then FSMs M_1 and M_2 are equivalent if and only if

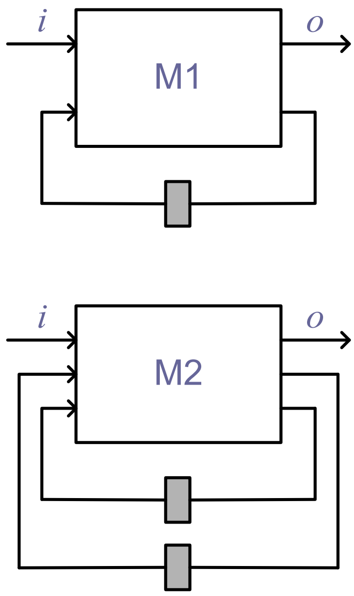
$$R(s) \rightarrow (\lambda_{1\times 2}(x,s) \equiv 0)$$

  is valid for all valuations on input variables x and state variables s.
  - This can be checked in constant time for BDD

Sequential Equivalence Checking

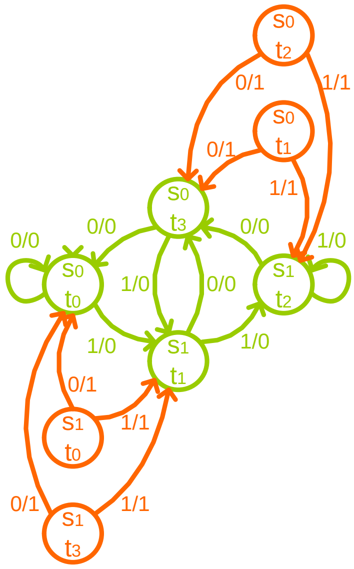
- Example
  - Are M1 and M2 equivalent ?

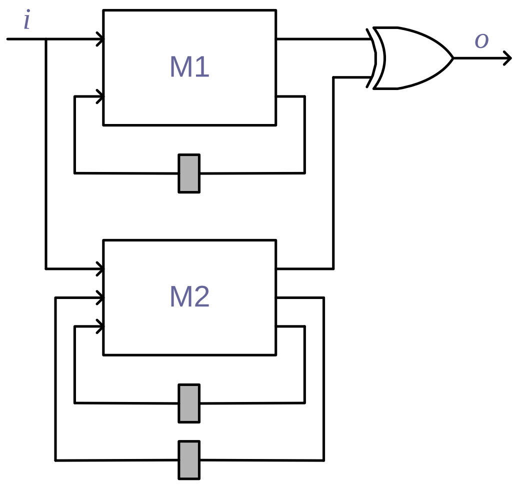




Sequential Equivalence Checking

- Example (cont'd)
  - Product FSM of M1 and M2

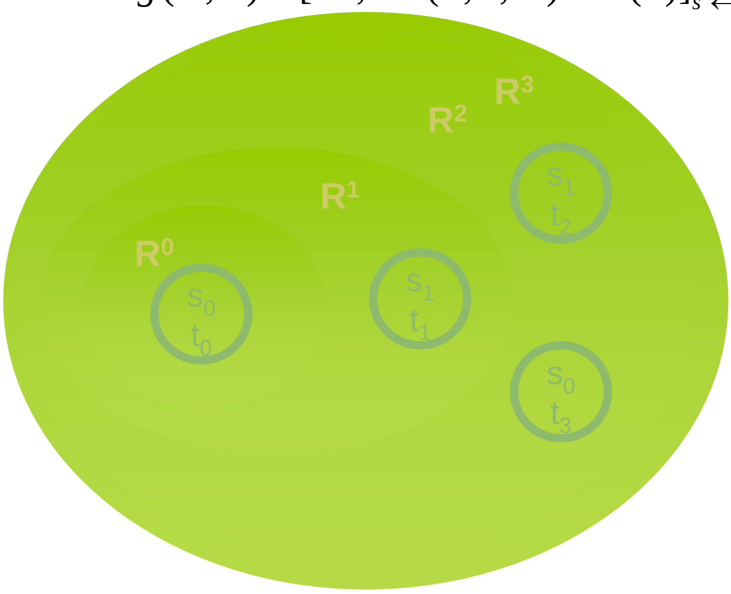


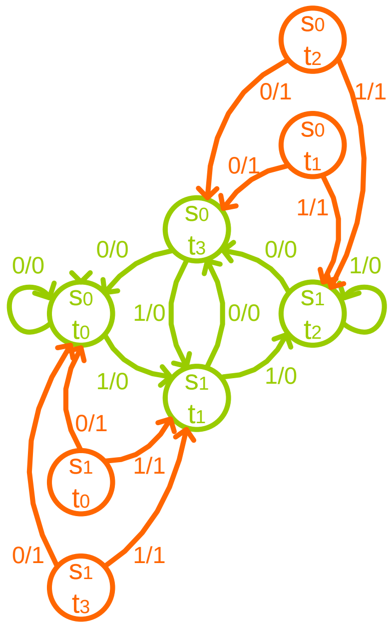


Sequential Equivalence Checking

- Example (cont'd)
  - Forward reachability analysis

$$Img(C,T) = [\exists \vec{x}, \vec{s}.T(\vec{x}, \vec{s}, \vec{s}\,') \wedge C(\vec{s})]_{\vec{s}\,' \leftarrow \vec{s}}$$

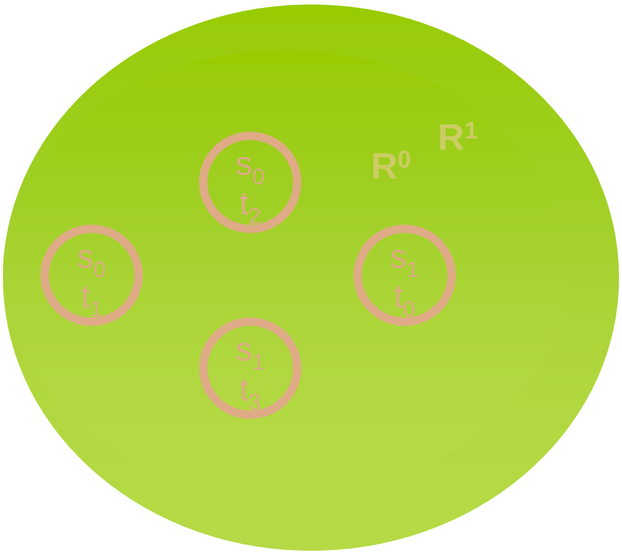


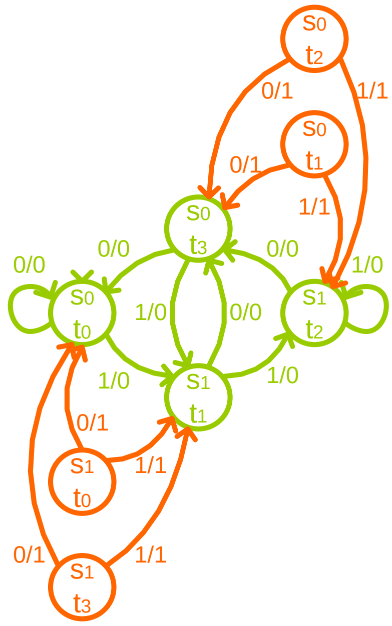


Sequential Equivalence Checking

- Example (cont'd)
  - Backward reachability analysis

    $PreImg(C,T) = \exists \vec{x}, \vec{s}\,'.T(\vec{x},\vec{s},\vec{s}\,') \wedge C(\vec{s}\,')$





Remarks on Sequential EC

- Industrial equivalence checkers almost exclusively use an combinational EC paradigm even for sequential EC
  - Sequential EC is too complex and can only be applied to design with a few hundred state bits
  - Structure similarity should be identified to simplify sequential EC
- Besides sequential equivalence checking, reachability analysis is useful in sequential circuit optimization
  - In sequential optimization, unreachable states can be used as sequential don't cares to optimize a sequential circuit

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
- Property checking
  - Safety property checking

Model Checking

- A specific model-checking problem is defined by

more detailed

$$M \models \varphi$$

more abstract

"implementation"
(system model)

"specification"
(system property)

"satisfies", "implements", "refines"
(satisfaction relation)

Model Checking

- M |= φ
  - Check if system model M satisfies a system property φ
  - System model M is described with a state transition system
    - finite state or infinite state
  - Temporal property φ can be described with three orthogonal choices:
    1. operational vs. declarative: automata vs. logic
    2. may vs. must: branching vs. linear time
    3. prohibiting bad vs. desiring good behavior: safety vs. liveness

  Different choices lead to different model checking problems.

Property Checking

- Safety property:
  Something "bad" will never happen
  - Safety property violation always has a finite witness
    - if something bad happens on an infinite run, then it happens already on some finite prefix
  - Example
    - Two processes cannot be in their critical sections simultaneously

- Liveness property:
  Something "good" will eventually happen
  - Liveness property violation never has a finite witness
    - no matter what happens along a finite run, something good could still happen later
  - Example
    - Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually

For finite state systems, liveness can be converted to safety!

Safety Property Checking

- Safety property checking can be formulated as a reachability problem
  - Are bad states reachable from good states?

- Sequential equivalence checking can be considered as one kind of safety property checking
  - M : product machine
  - φ : all states reachable from initial states has output 0

Model Checking

- Data structure evolution
  - State graph (late 70s-80s)
    - Problem size ~10^4 states
  - BDD (late 80s-90s)
    - Problem size ~10^{20} states
    - Critical resource: memory
  - SAT (late 90s-)
    - GRASP, SATO, chaff, berkmin
    - Problem size ~10^{100} (?) states
    - Critical resource: CPU time

Remarks on Model Checking

- Model checking is a very rich subject developed since early 1980's

- It is a variation of mathematical logic and is concerned with automatic temporal reasoning

- Reference

  M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.