Decision Procedures and Hardware Synthesis

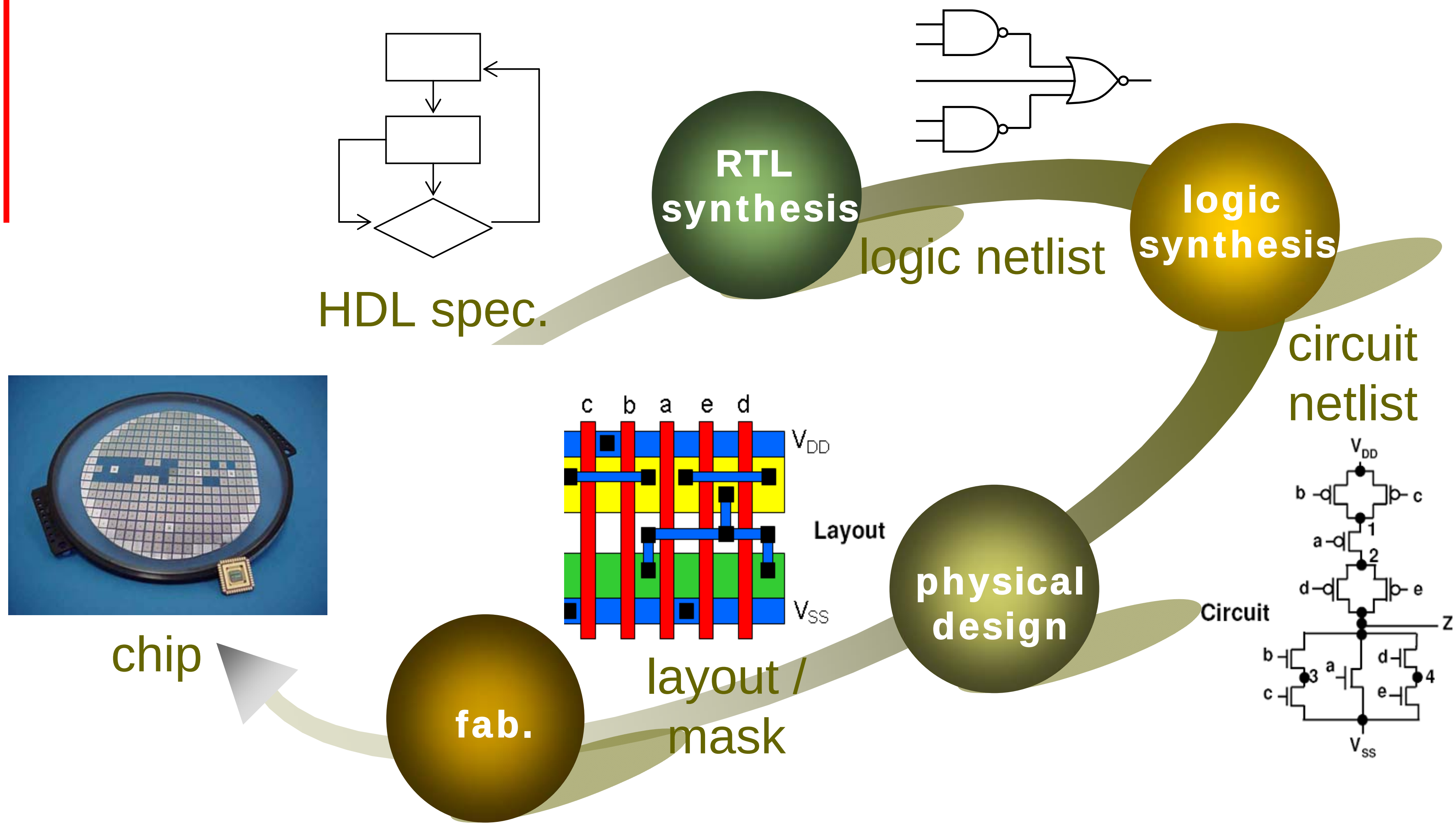
Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

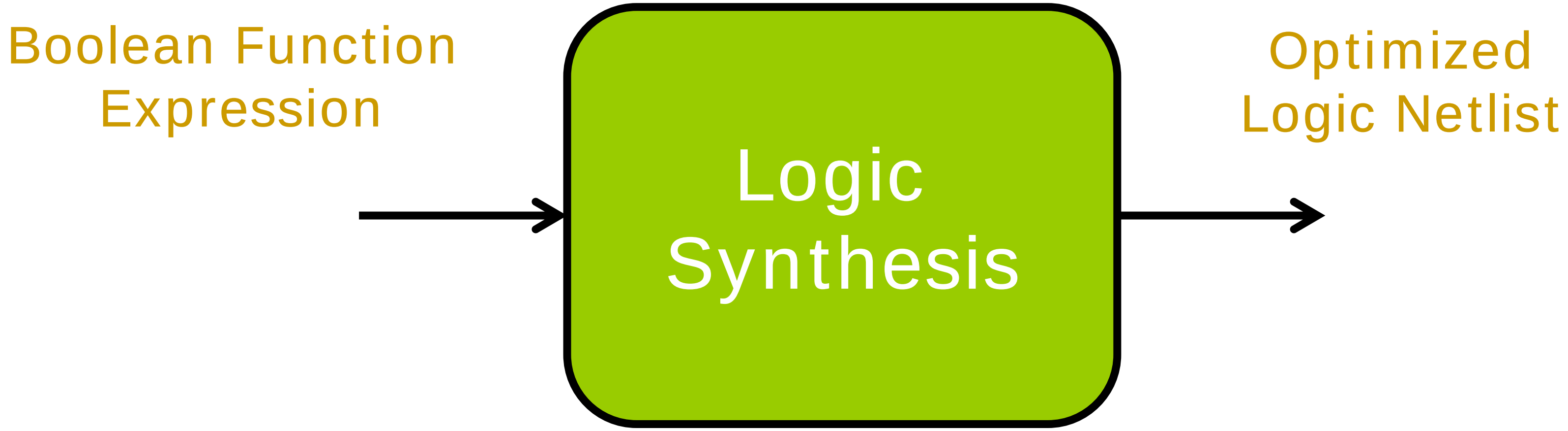
Outline

- Logic synthesis
- Boolean function representation
- Satisfiability and logic synthesis
  - Functional dependency
  - Functional bi-decomposition
- Quantified satisfiability and logic synthesis
  - Boolean matching
  - Boolean relation determinization

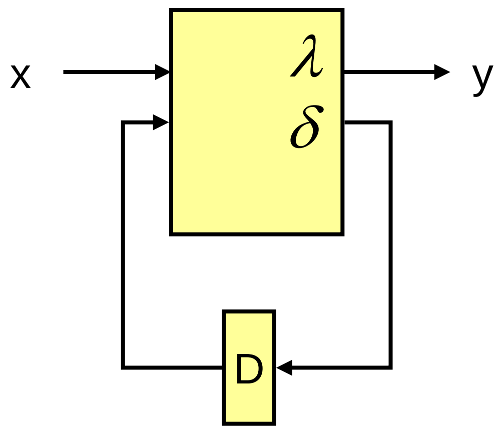
IC Design Flow



Logic Synthesis

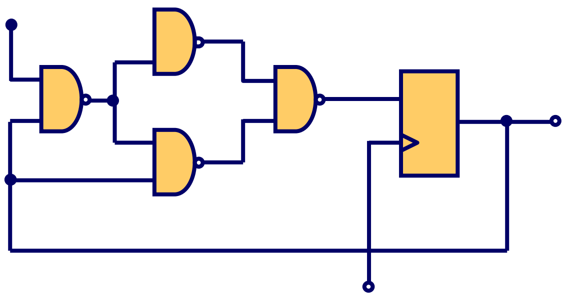


Logic Synthesis



Given: Functional description of finite-state machine F(Q,X,Y,δ,λ) where:

Q: Set of internal states
X: Input alphabet
Y: Output alphabet
δ: X x Q $\rightarrow$ Q (next state *function*)
λ: X x Q $\rightarrow$ Y (output *function*)



Target: Circuit C(G, W) where:
G: set of circuit components g $\in$ {gates, FFs, etc.}
W: set of wires connecting G

Boolean Function Representation

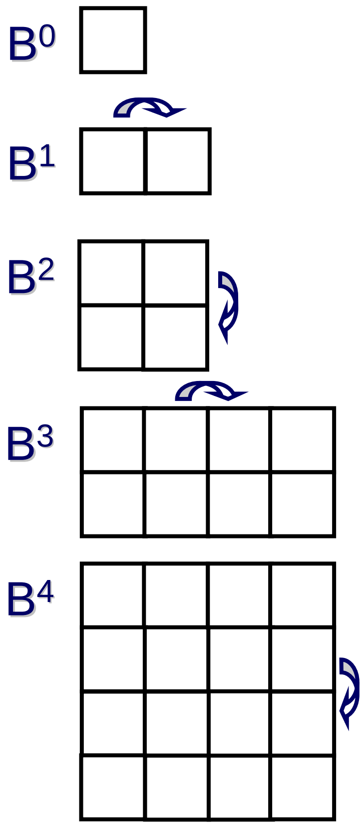
- Logic synthesis translates Boolean functions into circuits

- We need representations of Boolean functions for two reasons:
  - to represent and manipulate the actual circuit that we are implementing
  - to facilitate *Boolean reasoning*

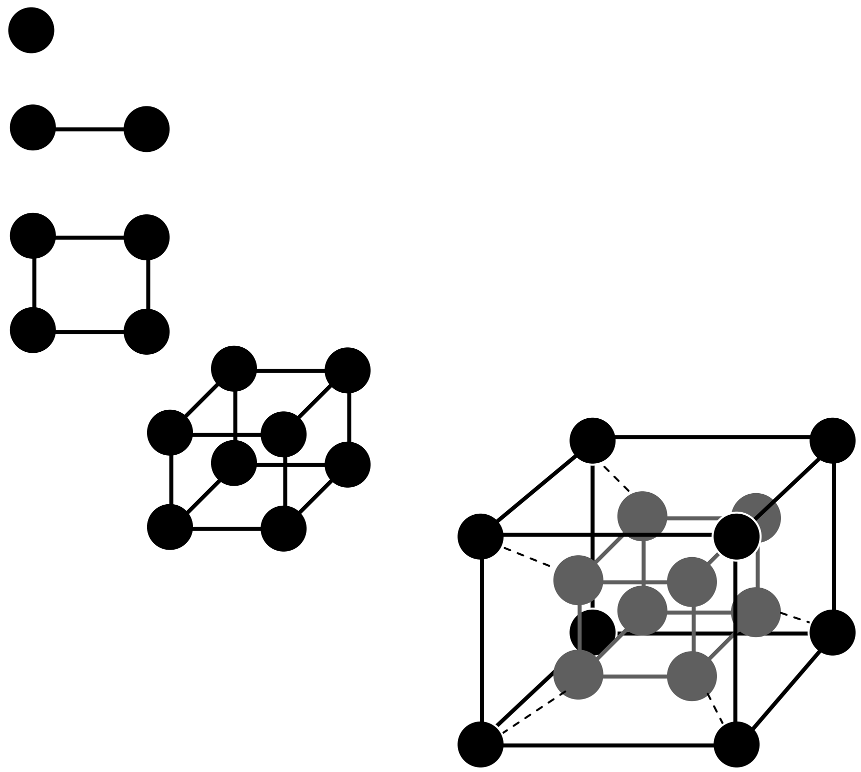
Boolean Space

- $B = \{0,1\}$
- $B^2 = \{0,1\}\times\{0,1\} = \{00, 01, 10, 11\}$

Karnaugh Maps:



Boolean Lattices:



Boolean Function

- A Boolean function f over input variables: $x_1, x_2, \ldots, x_m$, is a mapping f: $\mathbf{B}^m \rightarrow Y$, where $\mathbf{B} = \{0,1\}$ and $Y = \{0,1,d\}$
  - E.g.
  - The output value of $f(x_1, x_2, x_3)$, say, partitions $\mathbf{B}^m$ into three sets:
    - on-set ($f = 1$)
      - E.g. {010, 011, 110, 111} (characteristic function $f^1 = x_2$)
    - off-set ($f = 0$)
      - E.g. {100, 101} (characteristic function $f^0 = x_1 \neg x_2$)
    - don't-care set ($f = d$)
      - E.g. {000, 001} (characteristic function $f^d = \neg x_1 \neg x_2$)

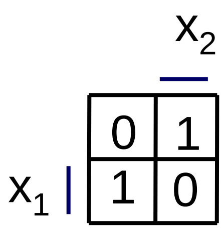
- *f* is an incompletely specified function if the don't-care set is nonempty. Otherwise, *f* is a completely specified function
  - Unless otherwise said, a Boolean function is meant to be completely specified

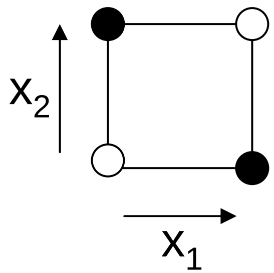
Boolean Function

- A Boolean function f: $\mathbf{B}^n \rightarrow \mathbf{B}$ over variables $x_1,\ldots,x_n$ maps each Boolean valuation (truth assignment) in $\mathbf{B}^n$ to 0 or 1

Example

$f(x_1,x_2)$ with $f(0,0) = 0$, $f(0,1) = 1$, $f(1,0) = 1$, $f(1,1) = 0$

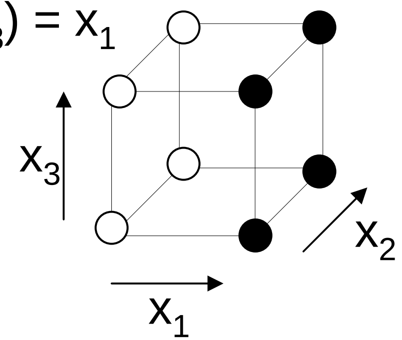




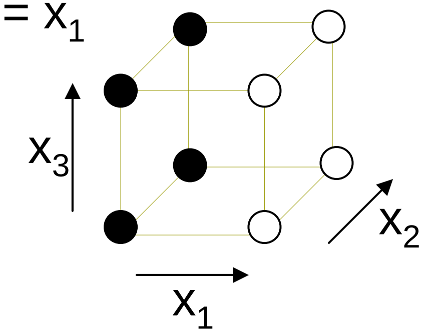
Boolean Function

- **Onset** of f, denoted as f^1, is $f^1 = \{v \in \mathbf{B}^n \mid f(v)=1\}$
  - If $f^1 = \mathbf{B}^n$, f is a tautology
- **Offset** of f, denoted as f^0, is $f^0 = \{v \in \mathbf{B}^n \mid f(v)=0\}$
  - If $f^0 = \mathbf{B}^n$, f is unsatisfiable. Otherwise, f is satisfiable.
- f^1 and f^0 are sets, not functions!
- Boolean functions f and g are equivalent if $\forall v \in \mathbf{B}^n.\ f(v) = g(v)$ where v is a truth assignment or Boolean valuation
- A literal is a Boolean variable x or its negation x' (or x, $\neg x$) in a Boolean formula

$f(x_1, x_2, x_3) = x_1$

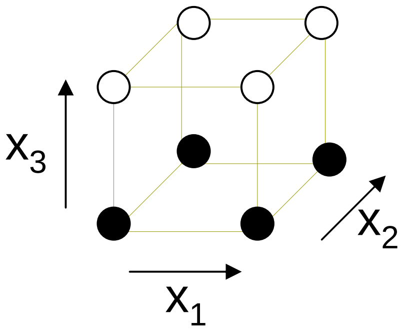


$f(x_1, x_2, x_3) = \overline{x_1}$



Boolean Function

- There are 2^n vertices in $\mathbf{B}^n$
- There are 2^{2^n} distinct Boolean functions
  - Each subset $f^1 \subseteq \mathbf{B}^n$ of vertices in $\mathbf{B}^n$ forms a distinct Boolean function f with onset f^1



$x_1x_2x_3$		f
0 0 0		1
0 0 1		0
0 1 0		1
0 1 1		0
1 0 0	$\Rightarrow$	1
1 0 1		0
1 1 0		1
1 1 1		0

Boolean Operations

Given two Boolean functions:

$f : \mathbf{B}^n \to \mathbf{B}$

$g : \mathbf{B}^n \to \mathbf{B}$

- $h = f \wedge g$ from AND operation is defined as
  $h^1 = f^1 \cap g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = f \vee g$ from OR operation is defined as
  $h^1 = f^1 \cup g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = \neg f$ from COMPLEMENT operation is defined as
  $h^1 = f^0$; $h^0 = f^1$

Cofactor and Quantification

Given a Boolean function:

$f: \mathbf{B}^n \rightarrow \mathbf{B}$, with the input variable $(x_1,x_2,\ldots,x_i,\ldots,x_n)$

- Positive cofactor on variable x_i
  $h = f_{xi}$ is defined as $h = f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Negative cofactor on variable x_i
  $h = f_{\neg xi}$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n)$
- Existential quantification over variable x_i
  $h = \exists x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \vee f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Universal quantification over variable x_i
  $h = \forall x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \wedge f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Boolean difference over variable x_i
  $h = \partial f/\partial x_i$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \oplus f(x_1,x_2,\ldots,1,\ldots,x_n)$

Boolean Function Representation

- Some common representations:
  - Truth table
  - Boolean formula
    - SOP (sum-of-products, or called disjunctive normal form, DNF)
    - POS (product-of-sums, or called conjunctive normal form, CNF)
  - BDD (binary decision diagram)
  - Boolean network (consists of nodes and wires)
    - Generic Boolean network
      - Network of nodes with generic functional representations or even subcircuits
    - Specialized Boolean network
      - Network of nodes with SOPs (PLAs)
      - And-Inv Graph (AIG)

- Why different representations?
  - Different representations have their own strengths and weaknesses (no single data structure is best for all applications)

Boolean Function Representation
Truth Table

- Truth table (function table for multi-valued functions):
  The truth table of a function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of $\mathbf{B}^n$.

  In other words the truth table lists all mintems

  Example: $f = a'b'c'd + a'b'cd + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' + abcd$

  The truth table representation is
  - impractical for large n
  - canonical

  If two functions are the equal, then their canonical representations are isomorphic.

	abcd	f
0	0000	0
1	0001	1
2	0010	0
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	0

	abcd	f
8	1000	0
9	1001	1
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	1
15	1111	1

Boolean Function Representation
Boolean Formula

- A Boolean formula is defined inductively as an expression with the following formation rules (syntax):

formula ::=		‘(‘ formula ‘)’	
	|	Boolean constant	(**true** or **false**)
	|	<Boolean variable>	
	|	formula “+” formula	(OR operator)
	|	formula “·” formula	(AND operator)
	|	$\neg$ formula	(complement)

Example

$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot (\neg x_1)))$

typically “·” is omitted and ‘(‘, ‘)’ are omitted when the operator priority is clear, e.g., $f = x_1\ x_2 + x_3 + x_4 \neg x_1$

Boolean Function Representation
Boolean Formula in SOP

- Any function can be represented as a sum-of-products (SOP), also called sum-of-cubes (a cube is a product term), or disjunctive normal form (DNF)

Example

φ = ab + a'c + bc

Boolean Function Representation
Boolean Formula in POS

- Any function can be represented as a product-of-sums (POS), also called conjunctive normal form (CNF)
  - Dual of the SOP representation

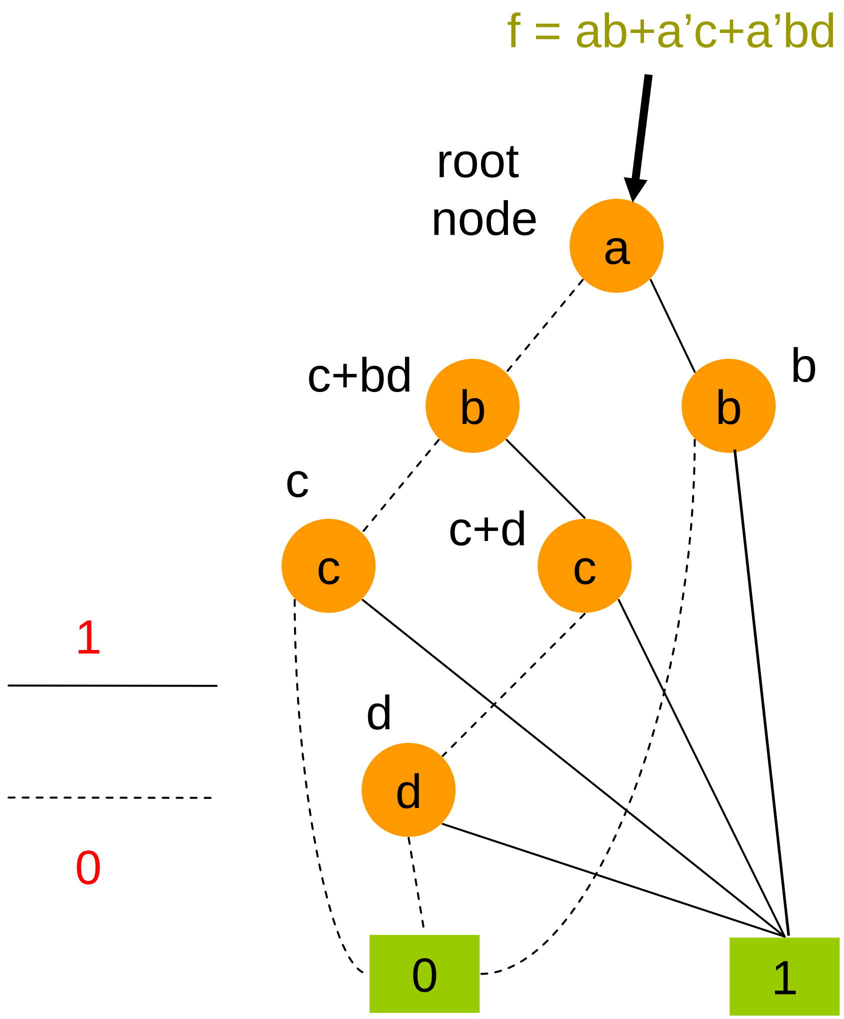
Example

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

- Exercise: Any Boolean function in POS can be converted to SOP using De Morgan's law and the distributive law, and vice versa

Boolean Function Representation
Binary Decision Diagram

- BDD – a graph representation of Boolean functions
  - A leaf node represents constant 0 or 1
  - A non-leaf node represents a decision node (multiplexer) controlled by some variable
  - Can make a BDD representation **canonical** by imposing the variable ordering and reduction criteria (ROBDD)



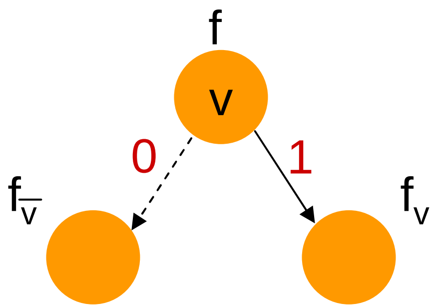
Boolean Function Representation
Binary Decision Diagram

- Any Boolean function *f* can be written in term of **Shannon expansion**

  $$f = v\, f_v + \neg v\, f_{\neg v}$$

  - Positive cofactor: $f_{xi} = f(x_1,\ldots,x_i=1,\ldots, x_n)$
  - Negative cofactor: $f_{\neg xi} = f(x_1,\ldots,x_i=0,\ldots, x_n)$

- BDD is a compressed Shannon cofactor tree:
  - The two children of a node with function *f* controlled by variable *v* represent two sub-functions f_v and $f_{\neg v}$

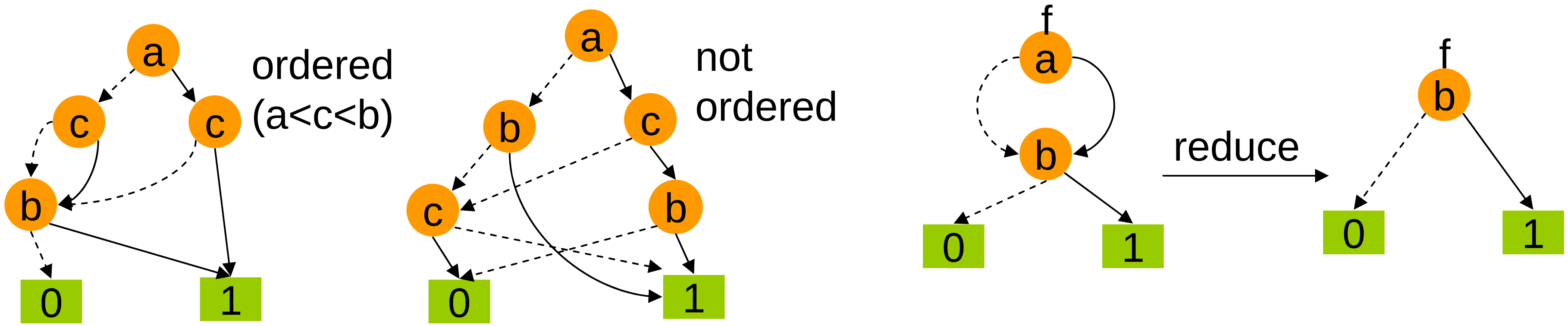


Boolean Function Representation
Binary Decision Diagram

- Reduced and ordered BDD (ROBDD) is a **canonical** Boolean function representation
  - Ordered:
    - cofactor variables are in the same order along all paths

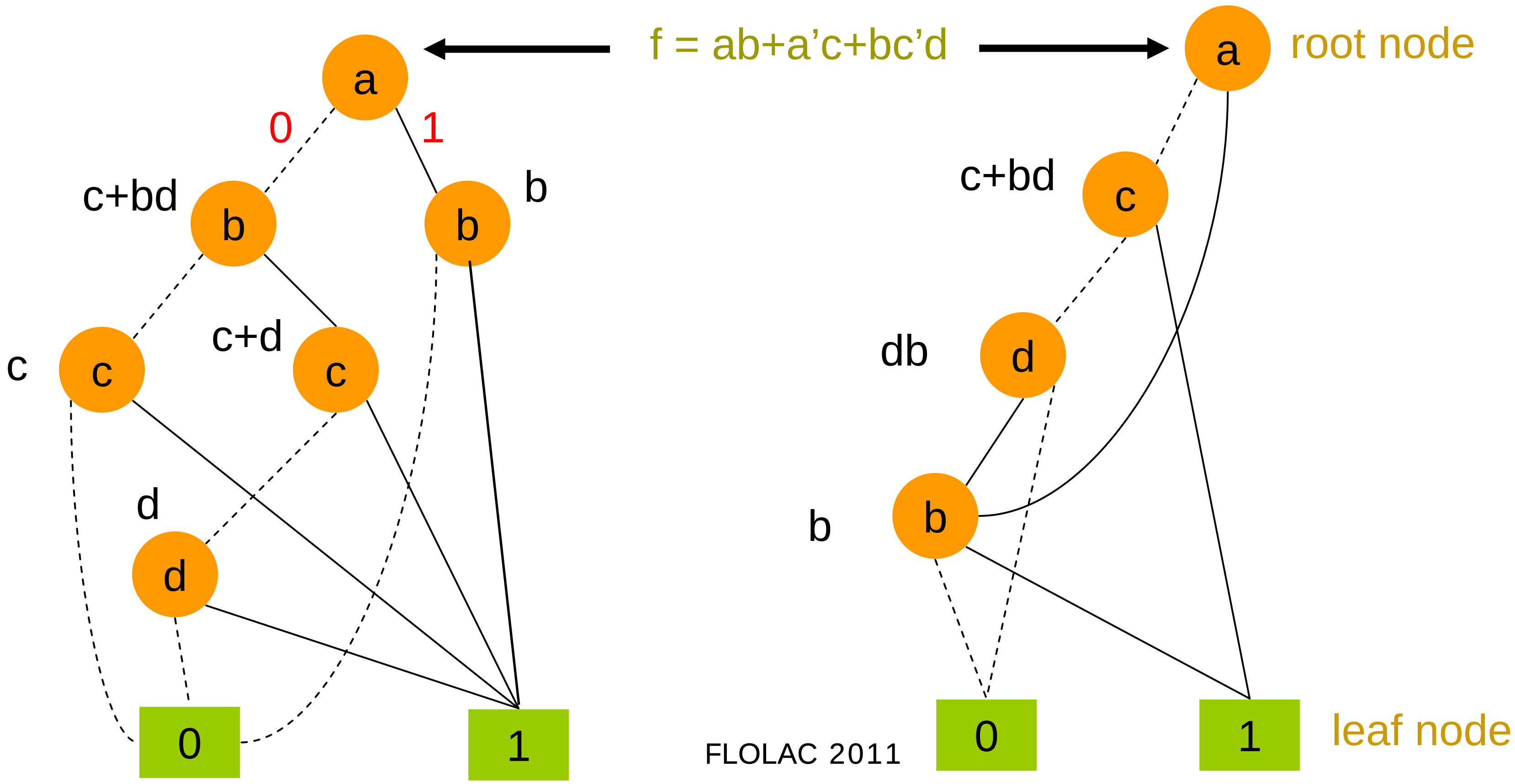
      $$x_{i_1} < x_{i_2} < x_{i_3} < \dots < x_{i_n}$$
  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged

    These two rules make any node in an ROBDD represent a distinct logic function



Boolean Function Representation
Binary Decision Diagram

- For a Boolean function,
  - ROBDD is unique with respect to a given variable ordering
  - Different orderings may result in different ROBDD structures



Boolean Function Representation
Boolean Network

- A Boolean network is a directed graph $C(G,N)$ where G are the gates and $N \subseteq (G \times G)$ are the directed edges (nets) connecting the gates.

  Some of the vertices are designated:
  Inputs: $I \subseteq G$
  Outputs: $O \subseteq G$
  $I \cap O = \emptyset$

  Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Function Representation
Boolean Network

- The fanin FI(g) of a gate g are the predecessor gates of g:
  $FI(g) = \{g' \mid (g',g) \in N\}$ (N: the set of nets)

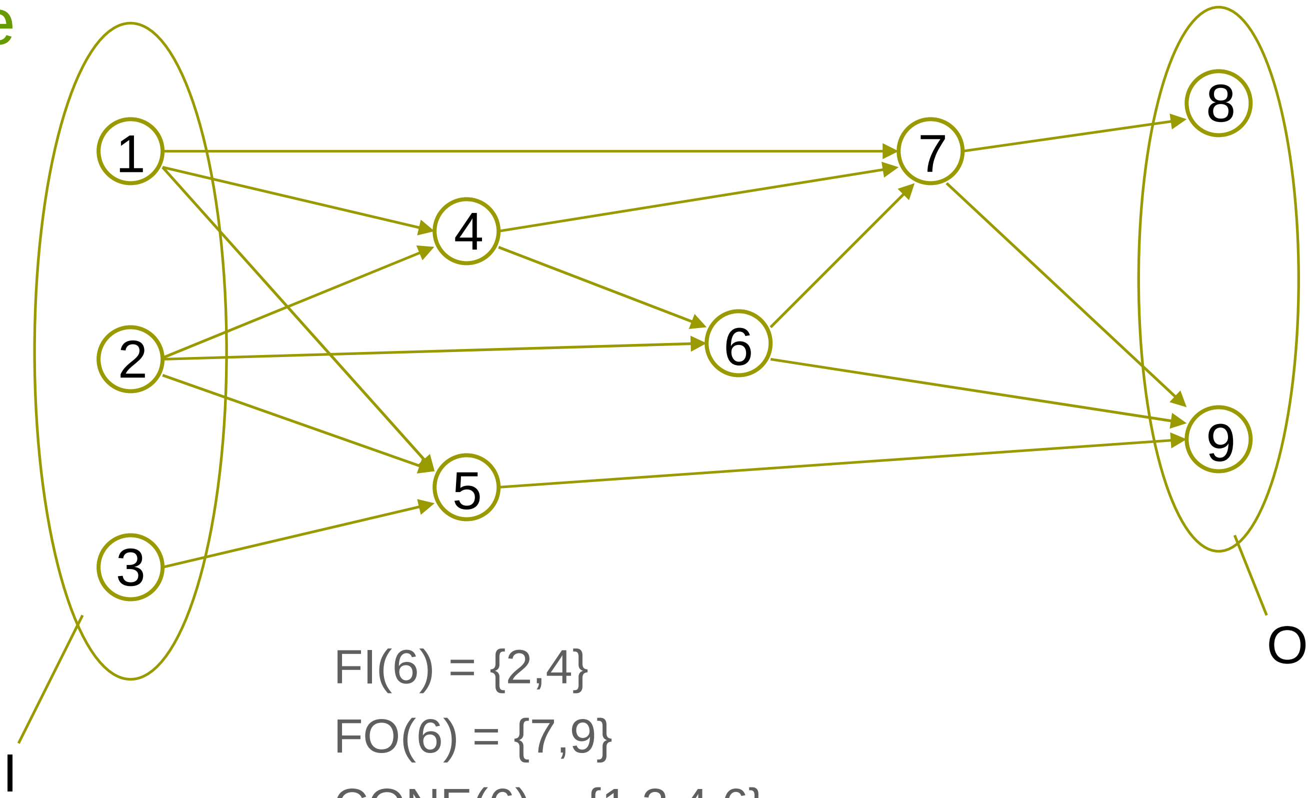
- The fanout FO(g) of a gate g are the successor gates of g:
  $FO(g) = \{g' \mid (g,g') \in N\}$

- The cone CONE(g) of a gate g is the transitive fanin (TFI) of g and g itself

- The support SUPPORT(g) of a gate g are all inputs in its cone:
  $SUPPORT(g) = CONE(g) \cap I$

Boolean Function Representation
Boolean Network

Example



FI(6) = {2,4}

FO(6) = {7,9}

CONE(6) = {1,2,4,6}

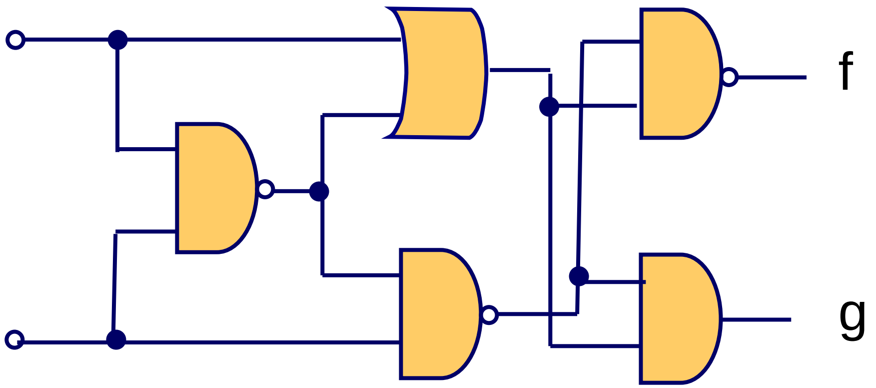
SUPPORT(6) = {1,2}

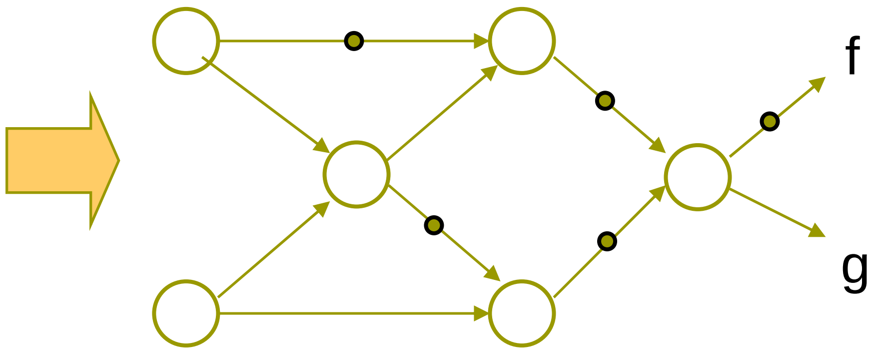
Every node may have its own function

Boolean Function Representation
And-Inverter Graph

- AND-INVERTER graphs (AIGs)
  vertices: 2-input AND gates
  edges: interconnects with (optional) dots representing INVs

- Hash table to identify and reuse structurally isomorphic circuits



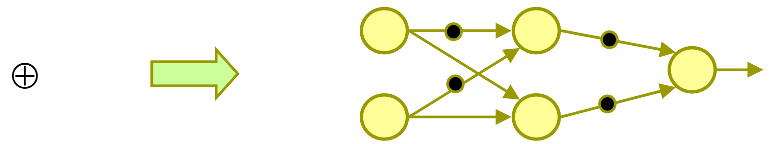


Boolean Function Representation

- Truth table
  - Canonical
  - Useful in representing small functions
- SOP
  - Useful in two-level logic optimization, and in representing local node functions in a Boolean network
- POS
  - Useful in SAT solving and Boolean reasoning
  - Rarely used in circuit synthesis (due to the asymmetric characteristics of NMOS and PMOS)
- ROBDD
  - Canonical
  - Useful in Boolean reasoning
- Boolean network
  - Useful in multi-level logic optimization
- AIG
  - Useful in multi-level logic optimization and Boolean reasoning

Circuit to CNF Conversion

- 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

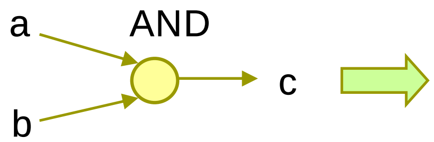


    - generated chess-board Karnaugh map
    - CNF (or DNF) formula has 2^{n-1} terms (exponential in #vars)

- 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

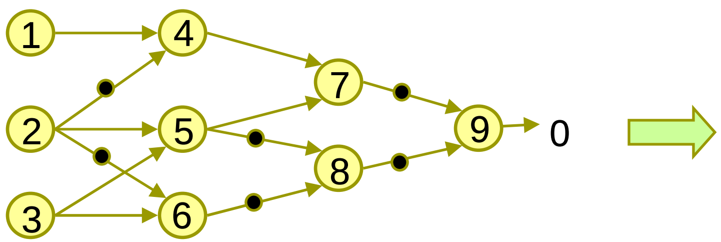
Circuit to CNF Conversion

- 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 Conversion

- 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

Propositional Satisfiability

Normal Forms

- A **literal** is a variable or its negation
- A **clause** (**cube**) is a disjunction (conjunction) of literals
- A **conjunctive normal form** (CNF) is a conjunction of clauses; a **disjunctive normal form** (DNF) is a disjunction of cubes

  - E.g.,

    CNF: $(a+\neg b+c)(a+\neg c)(b+d)(\neg a)$

    - $(\neg a)$ is a unit clause, d is a pure literal

    DNF: $a\neg bc + a\neg c + bd + \neg a$

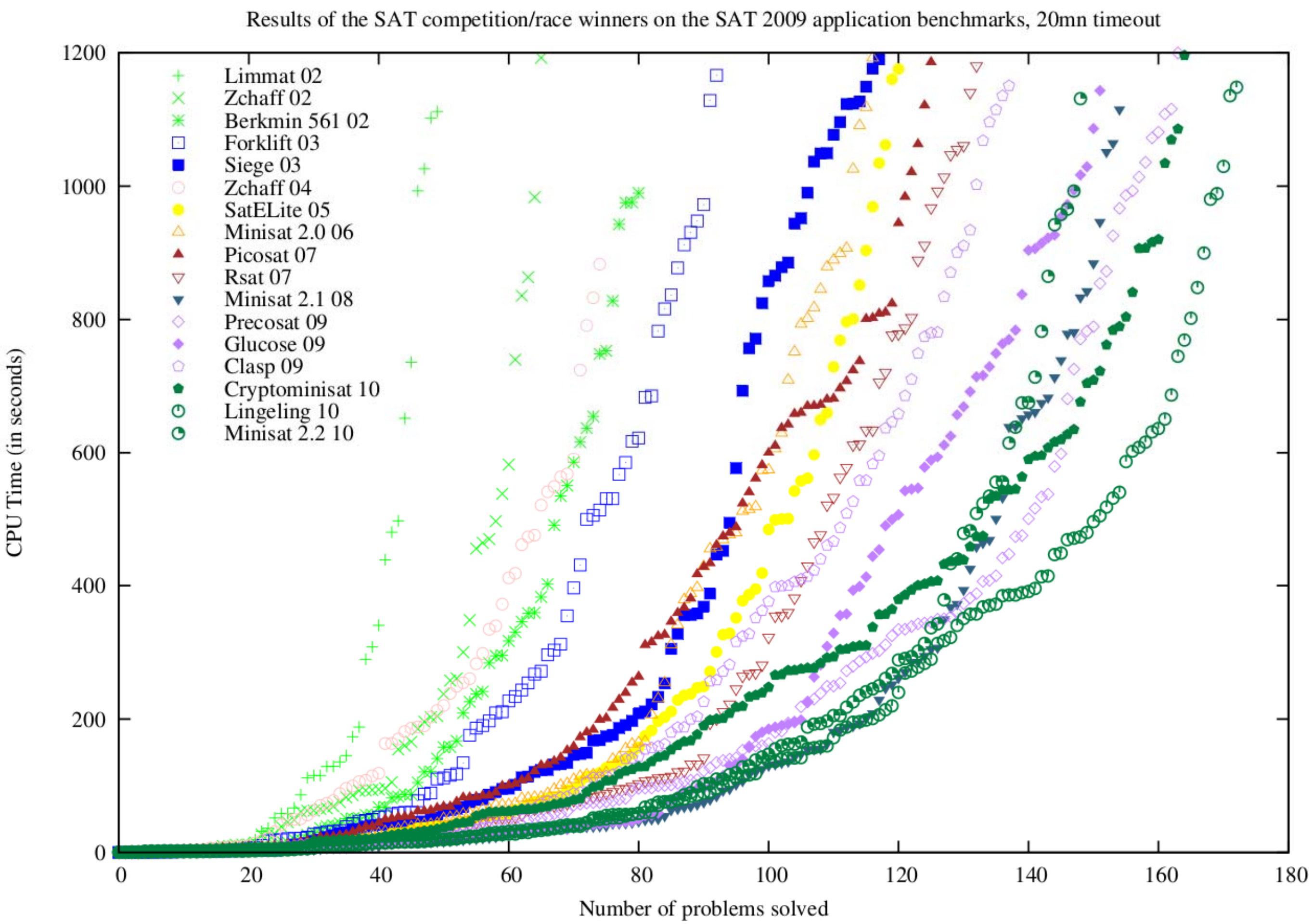
Satisfiability

- The **satisfiability** (SAT) problem asks whether a given CNF formula can be true under some assignment to the variables

- In theory, SAT is intractable
  - The first shown NP-complete problem [Cook, 1971]

- In practice, modern SAT solvers work 'mysteriously' well on application CNFs with ~100,000 variables and ~1,000,000 clauses
  - It enables various applications, and inspires QBF and SMT (Satisfiability Modulo Theories) solver development

SAT Competition



http://www.satcompetition.org/PoS11/

SAT Solving

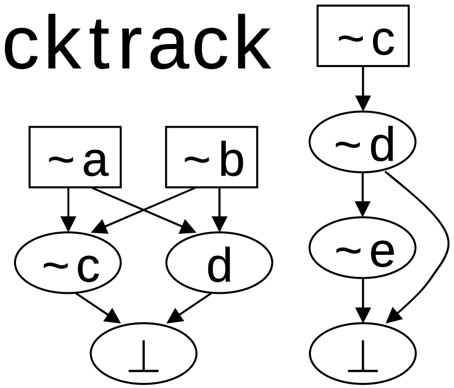
- Ingredients of modern SAT solvers:
  - DPLL-style search
    - [Davis, Putnam, Logemann, Loveland, 1962]
  - Conflict-driven clause learning (CDCL)
    - [Marques-Silva, Sakallah, 1996 (GRASP)]
  - Boolean constraint propagation (BCP) with two-literal watch
    - [Moskewicz, Modigan, Zhao, Zhang, Malik, 2001 (Chaff)]
  - Decision heuristics using variable activity
    - [Moskewicz, Modigan, Zhao, Zhang, Malik, 2001 (Chaff)]
  - Restart
  - Preprocessing
  - Support for incremental solving
    - [Een, Sorensson, 2003 (MiniSat)]

Pre-Modern SAT Procedure

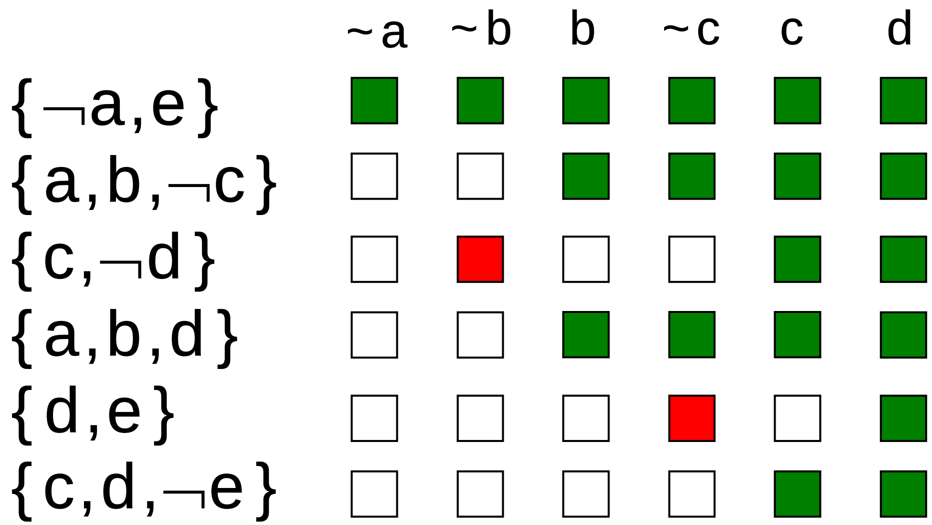
```
Algorithm DPLL(Φ)
{
  while there is a unit clause {l} in Φ
     Φ = BCP(Φ, l);
  while there is a pure literal l in Φ
     Φ = assign(Φ, l);
  if all clauses of Φ satisfied   return true;
  if Φ has a conflicting clause   return false;
  l := choose_literal(Φ);
  return DPLL(assign(Φ,¬l)) ∨ DPLL(assign(Φ,l));
}
```

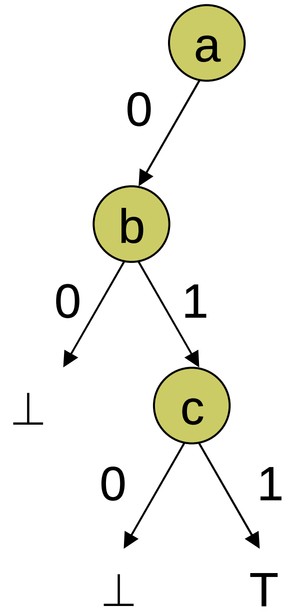
DPLL Procedure

- Chorological backtrack



- E.g.





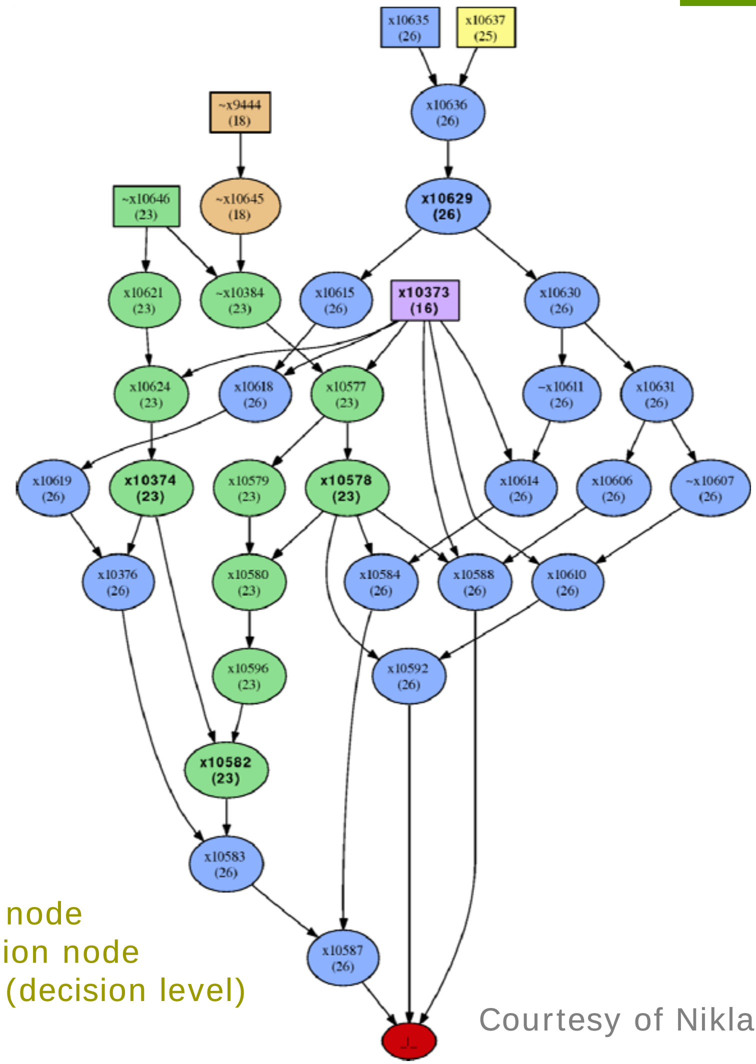
Modern SAT Procedure

```
Algorithm CDCL(Φ)
{
  while(1)
    while there is a unit clause {l} in Φ
        Φ = BCP(Φ, l);
    while there is a pure literal l in Φ
        Φ = assign(Φ, l);
    if Φ contains no conflicting clause
      if all clauses of Φ are satisfied   return true;
      l := choose_literal(Φ);
      assign(Φ,l);
    else
      if conflict at top decision level   return false;
      analyze_conflict();
      undo assignments;
      Φ := add_conflict_clause(Φ);
}
```

Conflict Analysis & Clause Learning

- There can be many learnt clauses from a conflict
- Clause learning admits non-chorological backtrack

- E.g.,
  {¬x10587, ¬x10588, ¬x10592}
  …
  {¬x10374, ¬x10582, ¬x10578, ¬x10373, ¬x10629}
  …
  {x10646, x9444, ¬x10373, ¬x10635, ¬x10637}



Box: decision node
Oval: implication node
Inside: literal (decision level)

Courtesy of Niklas Een

Clause Learning as Resolution

- **Resolution** of two clauses $C_1 \lor x$ and $C_2 \lor \neg x$:

$$\frac{C_1 \lor x \qquad C_2 \lor \neg x}{C_1 \lor C_2}$$

  where x is the **pivot variable** and $C_1 \lor C_2$ is the **resolvant**, i.e., $C_1 \lor C_2 = \exists x.(C_1 \lor x)(C_2 \lor \neg x)$

- A learnt clause can be obtained from a sequence of resolution steps
  - Exercise:

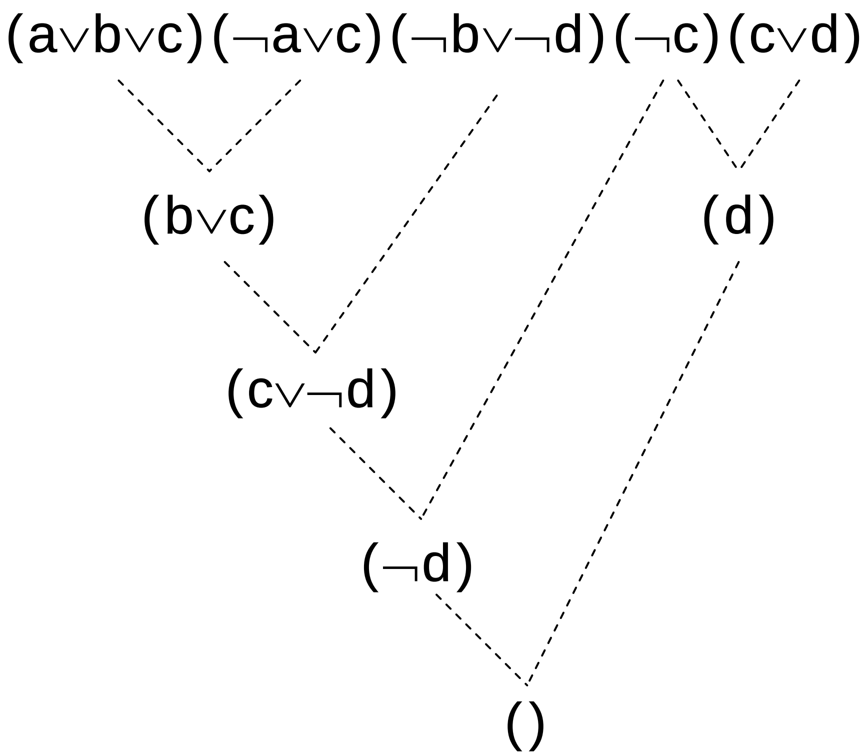
    Find a resolution sequence leading to the learnt clause

    $\{\neg x10374, \neg x10582, \neg x10578, \neg x10373, \neg x10629\}$

    in the previous slides

Resolution

- Resolution is complete for SAT solving
  - A CNF formula is unsatisfiable if and only if there exists a resolution sequence leading to the empty clause
  - Example



SAT Certification

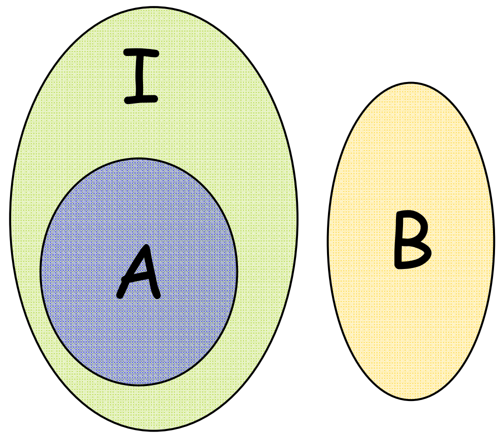
- True CNF
  - Satisfying assignment (model)
    - Verifiable in linear time

- False CNF
  - Resolution refutation
    - Potentially of exponential size

Craig Interpolation

- [Craig Interpolation Thm, 1957]
  If $A \wedge B$ is UNSAT for formulae A and B, there exists an interpolant I of A such that

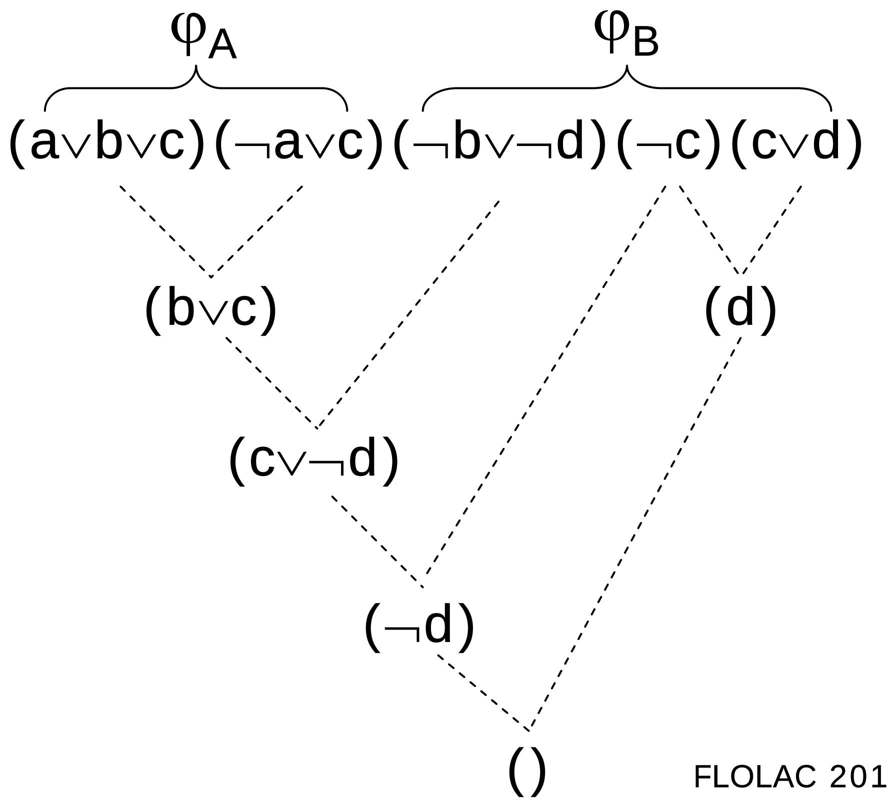
  1. $A \Rightarrow I$
  2. $I \wedge B$ is UNSAT
  3. I refers only to the common variables of A and B

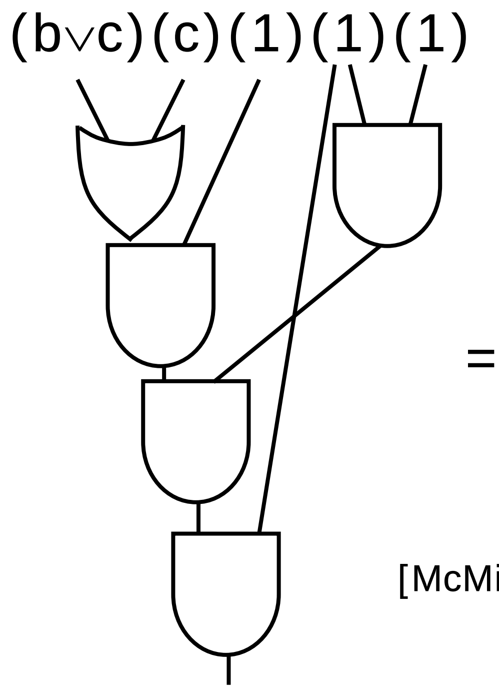


I is an abstraction of A

Interpolant and Resolution Proof

- SAT solver may produce the resolution proof of an UNSAT CNF φ
- For $\varphi = \varphi_A \wedge \varphi_B$ specified, the corresponding interpolant can be obtained in time linear in the resolution proof





$= (b \vee c)$

[McMillan, 2003]

Incremental SAT Solving

- To solve, in a row, multiple CNF formulae, which are similar except for a few clauses, can we reuse the learnt clauses?
  - What if adding a clause to φ?
  - What if deleting a clause from φ?

Incremental SAT Solving

- MiniSat API
  - void *addClause*(Vec<Lit> clause)
  - bool *solve*(Vec<Lit> assumps)
  - bool *readModel*(Var x) – *for SAT results*
  - bool *assumpUsed*(Lit p) – *for UNSAT results*

  - The method *solve()* treats the literals in assumps as unit clauses to be temporary assumed during the SAT-solving.
  - More clauses can be added after *solve()* returns, then incrementally another SAT-solving executed.

Courtesy of Niklas Een

SAT & Logic Synthesis

Functional Dependency

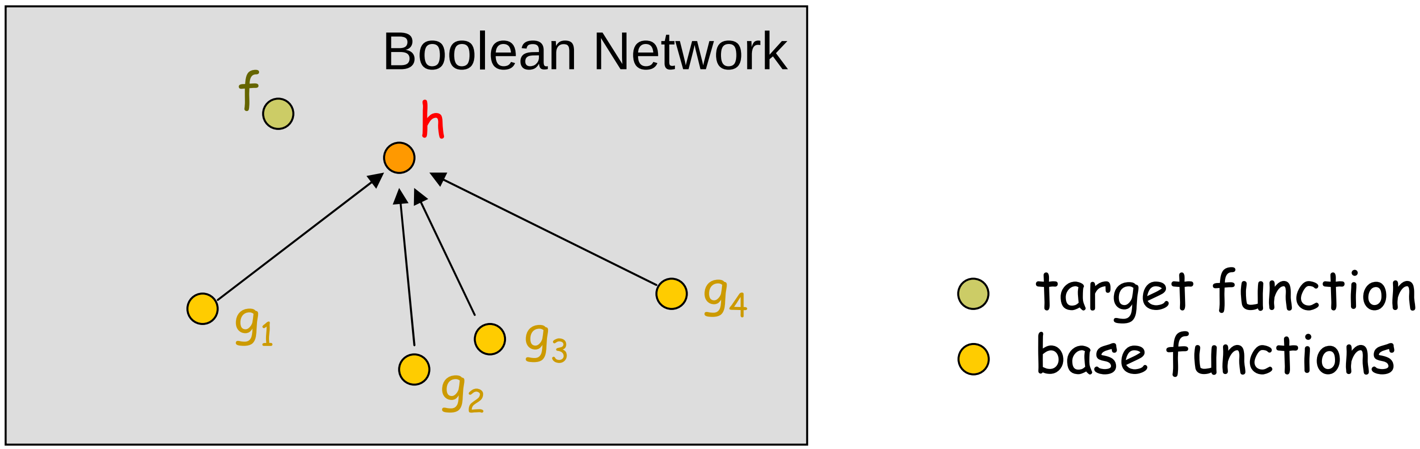
Functional Dependency

- $f(x)$ **functionally depends** on $g_1(x)$, $g_2(x)$, …, $g_m(x)$ if $f(x) = h(g_1(x), g_2(x), …, g_m(x))$, denoted $h(G(x))$
  - Under what condition can function f be expressed as some function h over a set $G=\{g_1,…,g_m\}$ of functions ?
  - h exists $\Leftrightarrow$ $\nexists a,b$ such that $f(a)\neq f(b)$ and $G(a)=G(b)$

i.e., G is more distinguishing than f

Motivation

- Applications of functional dependency
  - Resynthesis/rewiring
  - Redundant register removal
  - BDD minimization
  - Verification reduction
  - …

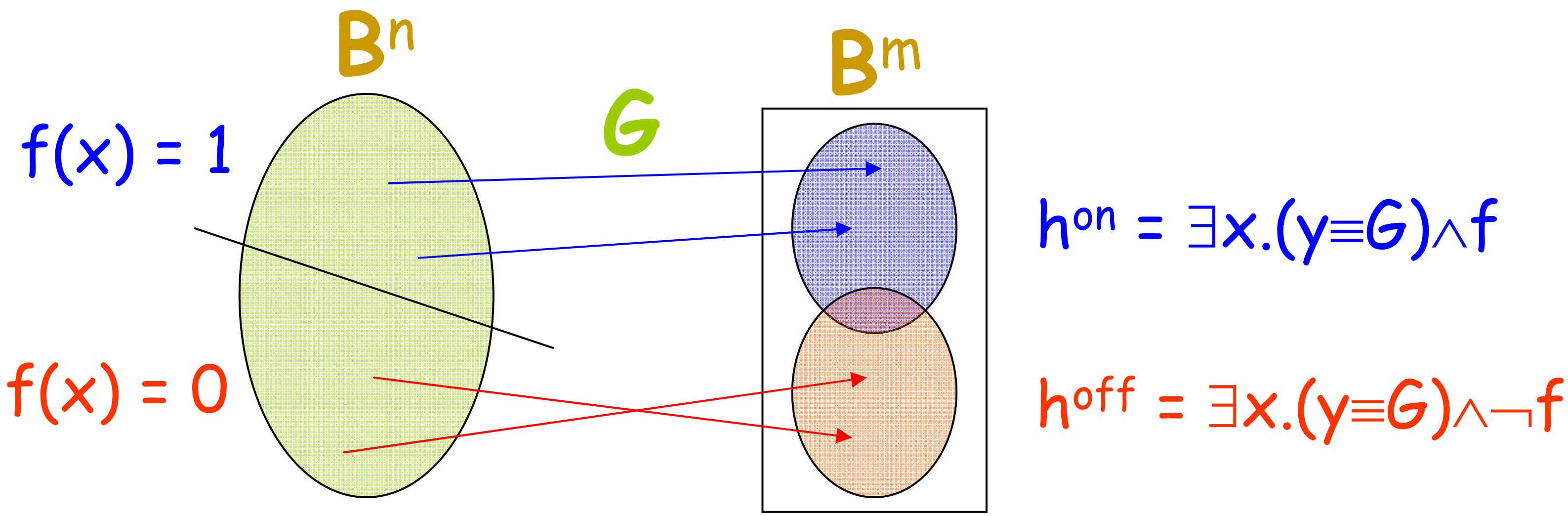


BDD-Based Computation

- BDD-based computation of h

  $h^{on} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 1, x \in \mathbf{B}^n\}$

  $h^{off} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 0, x \in \mathbf{B}^n\}$



BDD-Based Computation

- Pros
  - Exact computation of h^{on} and h^{off}
  - Better support for don't care minimization

- Cons
  - 2 image computations for every choice of G
  - Inefficient when $|G|$ is large or when there are many choices of G

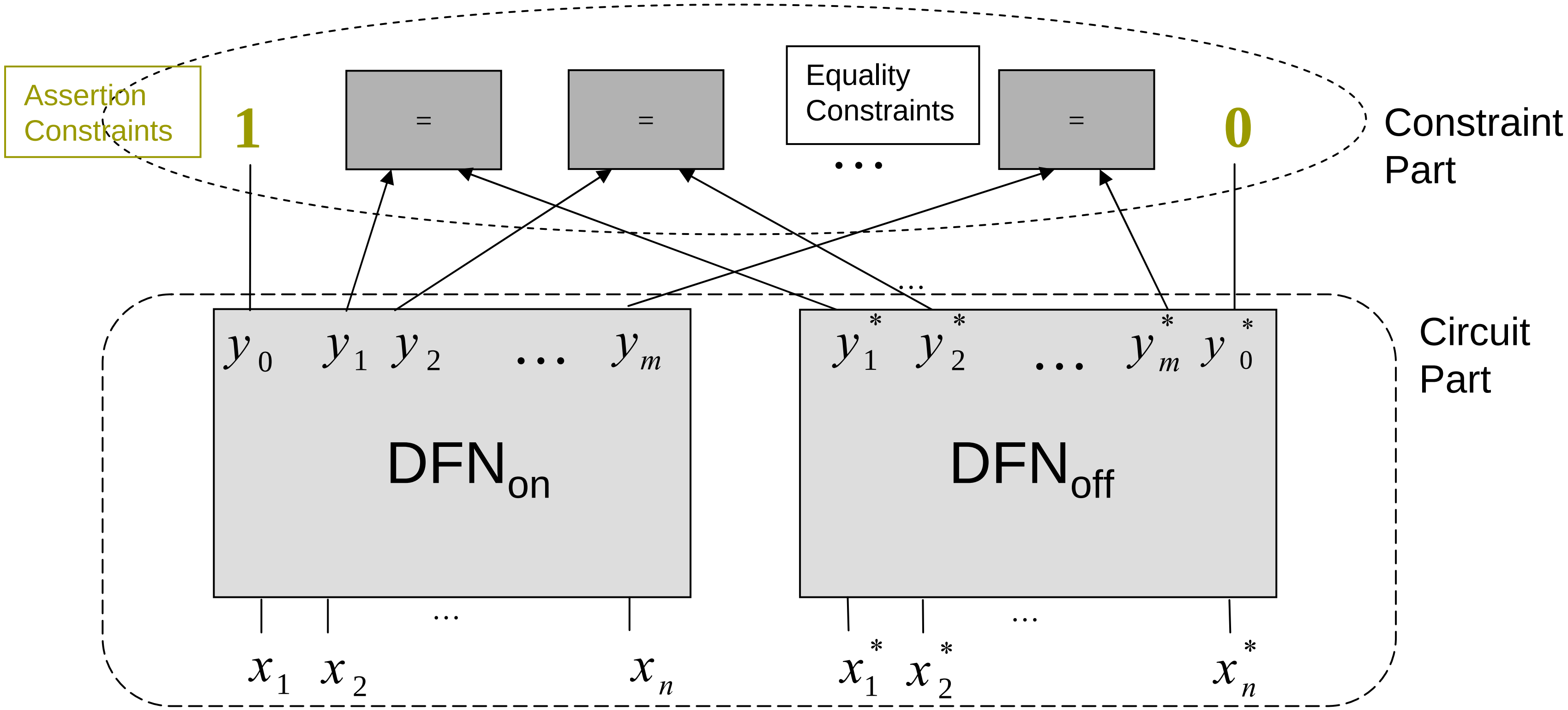
SAT-Based Computation

- h exists $\Leftrightarrow$
  $\nexists a,b$ such that $f(a) \neq f(b)$ and $G(a)=G(b)$,
  i.e., $(f(x) \not\equiv f(x^*)) \wedge (G(x) \equiv G(x^*))$ is UNSAT

- How to derive h? How to select G?

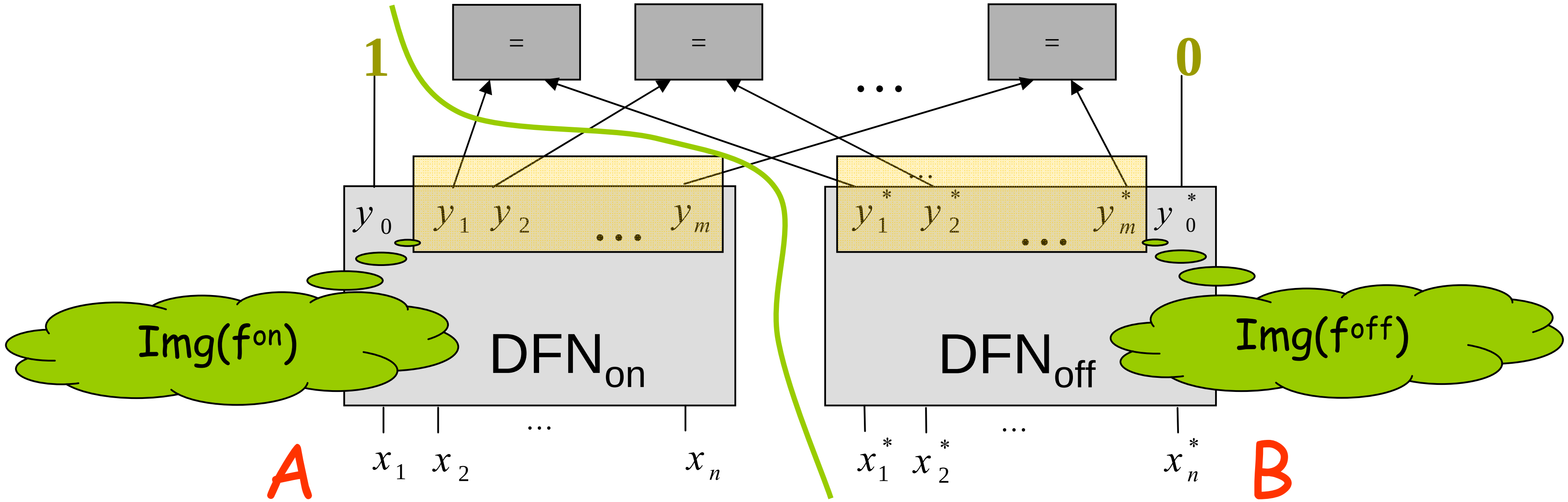
SAT-Based Computation

- $(f(x) \not\equiv f(x^*)) \wedge (G(x) \equiv G(x^*))$ is UNSAT



SAT-Based Computation

- Clause set A: C_{DFNon}, y_0
- Clause set B: C_{DFNoff}, $\neg y_0^*$, $(y_i \equiv y_i^*)$ for $i = 1,\ldots,m$
- I is an overapproximation of $Img(f^{on})$ and is disjoint from $Img(f^{off})$
- I only refers to $y_1,\ldots, y_m$
- Therefore, I corresponds to a feasible implementation of h



Incremental SAT Solving

- Controlled equality constraints
  $(y_i \equiv y_i^*) \rightarrow (\neg y_i \vee y_i^* \vee \alpha_i)(y_i \vee \neg y_i^* \vee \alpha_i)$
  with auxiliary variables α_i

  α_i = true $\Rightarrow$ i^{th} equality constraint is disabled

  - Fast switch between target and base functions by unit assumptions over control variables
  - Fast enumeration of different base functions
  - Share learned clauses

SAT vs. BDD

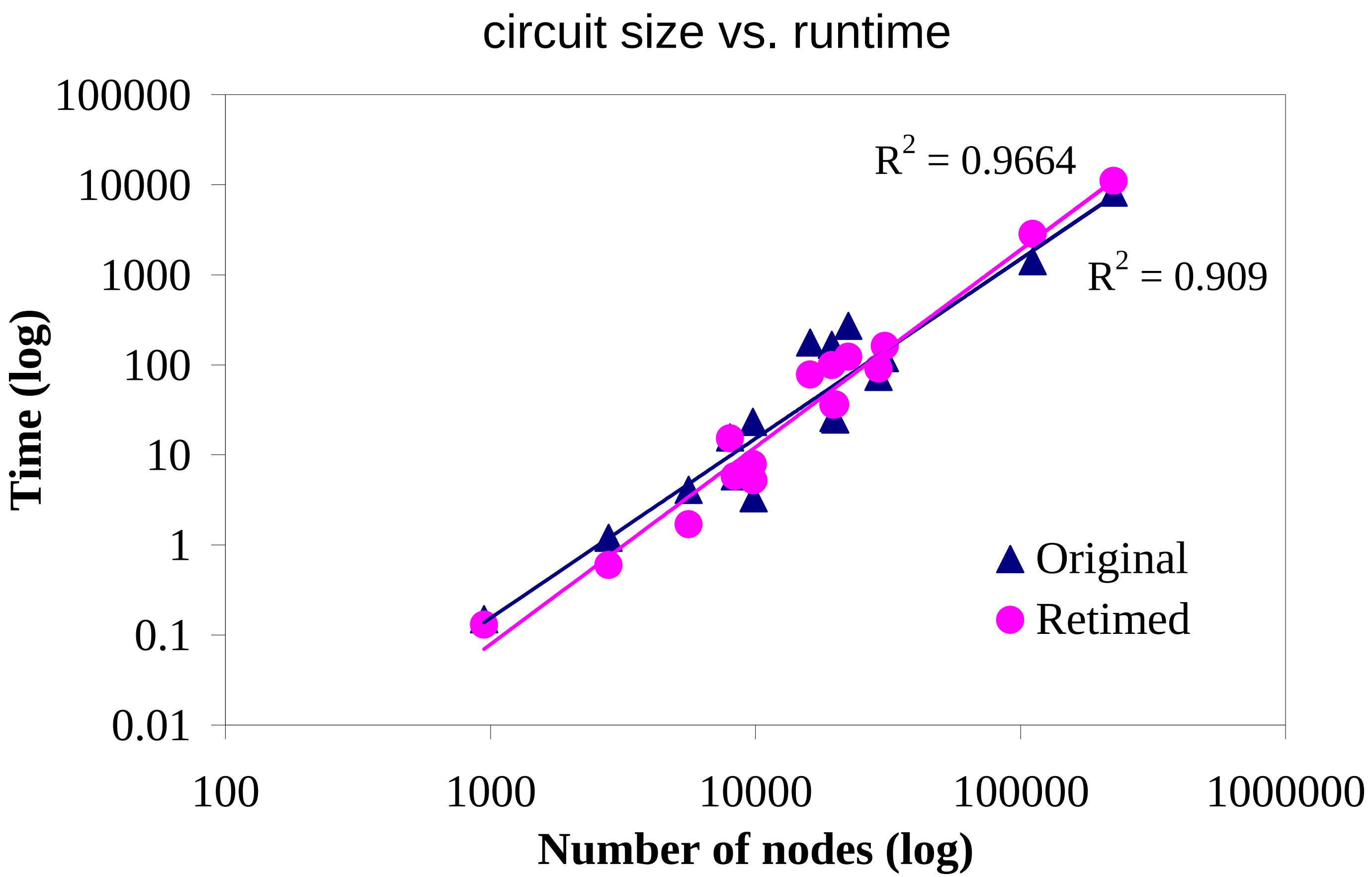
- SAT
  - Pros
    - Detect multiple choices of G automatically
    - Scalable to large $|G|$
    - Fast enumeration of different target functions f
    - Fast enumeration of different base functions G
  - Cons
    - Single feasible implementation of h
- BDD
  - Cons
    - Detect one choice of G at a time
    - Limited to small $|G|$
    - Slow enumeration of different target functions f
    - Slow enumeration of different base functions G
  - Pros
    - All possible implementations of h

Practical Evaluation

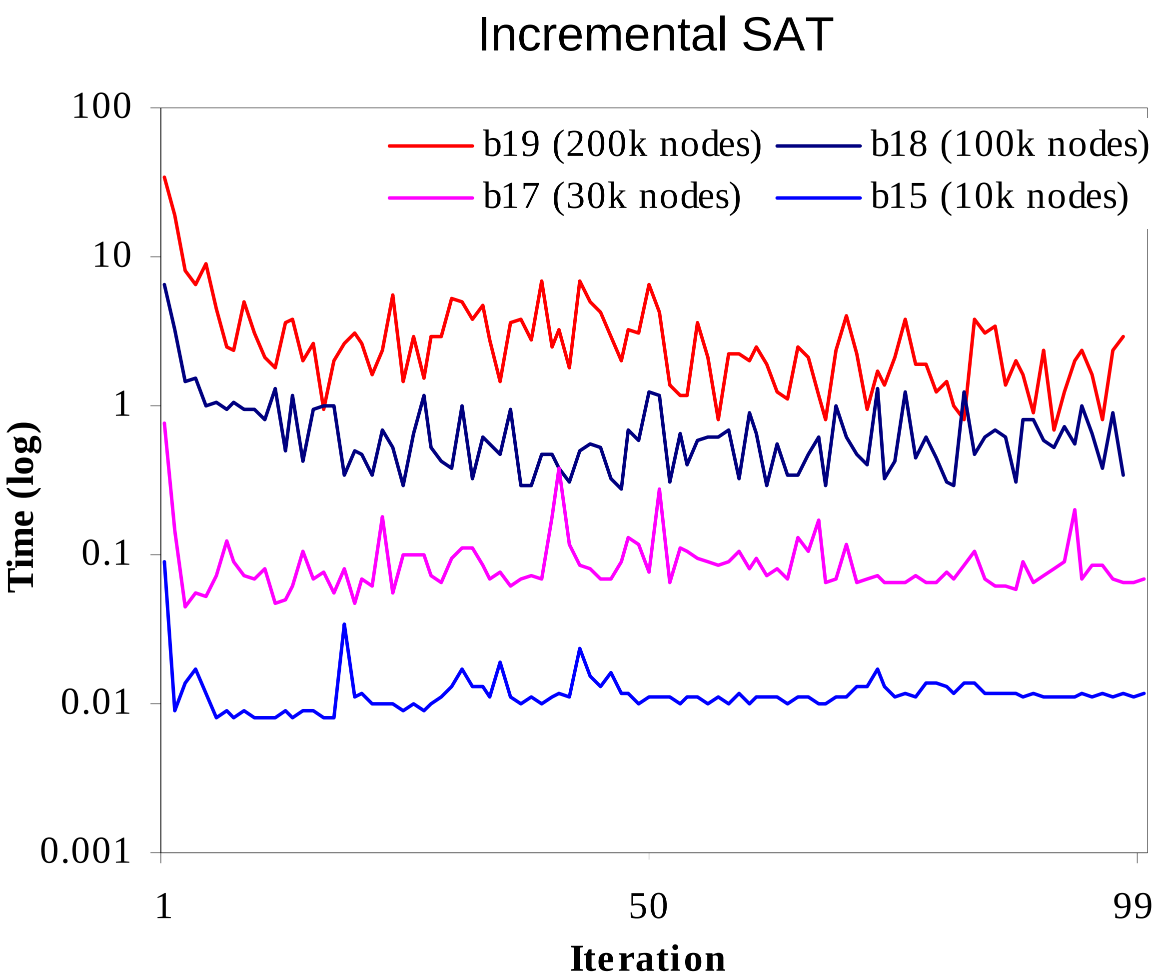
SAT vs. BDD

		Original			Retimed			SAT (original)		BDD (original)		SAT (retimed)		BDD (retimed)	
Circuit	#Nodes	#FF.	#Dep-S	#Dep-B	#FF.	#Dep-S	#Dep-B	Time	Mem	Time	Mem	Time	Mem	Time	Mem
s5378	2794	179	52	25	398	283	173	1.2	18	1.6	20	0.6	18	7	51
s9234.1	5597	211	46	x	459	301	201	4.1	19	x	x	1.7	19	194.6	149
s13207.1	8022	638	190	136	1930	802	x	15.6	22	31.4	78	15.3	22	x	x
s15850.1	9785	534	18	9	907	402	x	23.3	22	82.6	94	7.9	22	x	x
s35932	16065	1728	0	--	2026	1170	--	176.7	27	1117	164	78.1	27	--	--
s38417	22397	1636	95	--	5016	243	--	270.3	30	--	--	123.1	32	--	--
s38584	19407	1452	24	--	4350	2569	--	166.5	21	--	--	99.4	30	1117	164
b12	946	121	4	2	170	66	33	0.15	17	12.8	38	0.13	17	2.5	42
b14	9847	245	2	--	245	2	--	3.3	22	--	--	5.2	22	--	--
b15	8367	449	0	--	1134	793	--	5.8	22	--	--	5.8	22	--	--
b17	30777	1415	0	--	3967	2350	--	119.1	28	--	--	161.7	42	--	--
b18	111241	3320	5	--	9254	5723	--	1414	100	--	--	2842.6	100	--	--
b19	224624	6642	0	--	7164	337	--	8184.8	217	--	--	11040.6	234	--	--
b20	19682	490	4	--	1604	1167	--	25.7	28	--	--	36	30	--	--
b21	20027	490	4	--	1950	1434	--	24.6	29	--	--	36.3	31	--	--
b22	29162	735	6	--	3013	2217	--	73.4	36	--	--	90.6	37	--	--

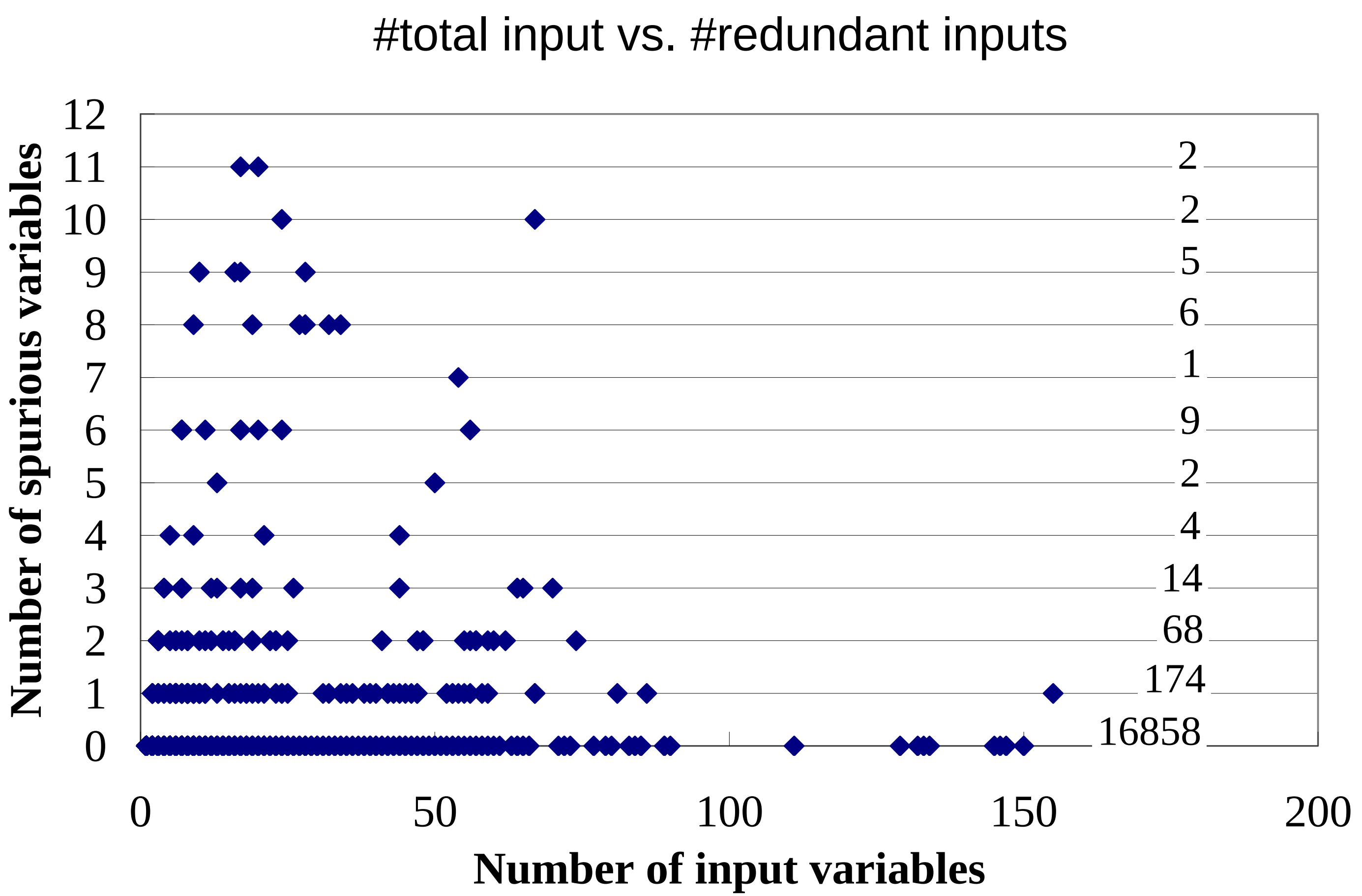
Practical Evaluation



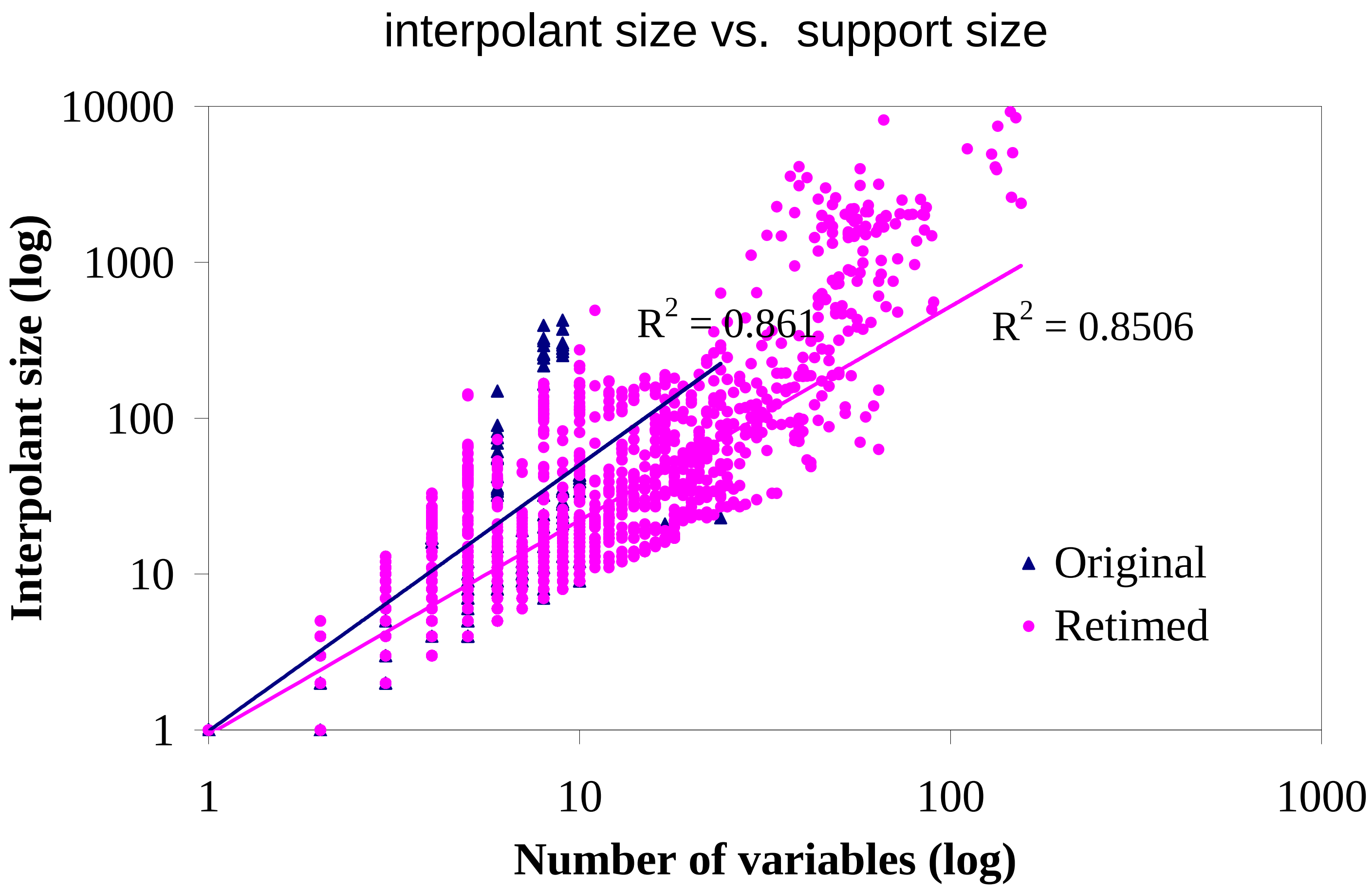
Practical Evaluation



Practical Evaluation



Practical Evaluation

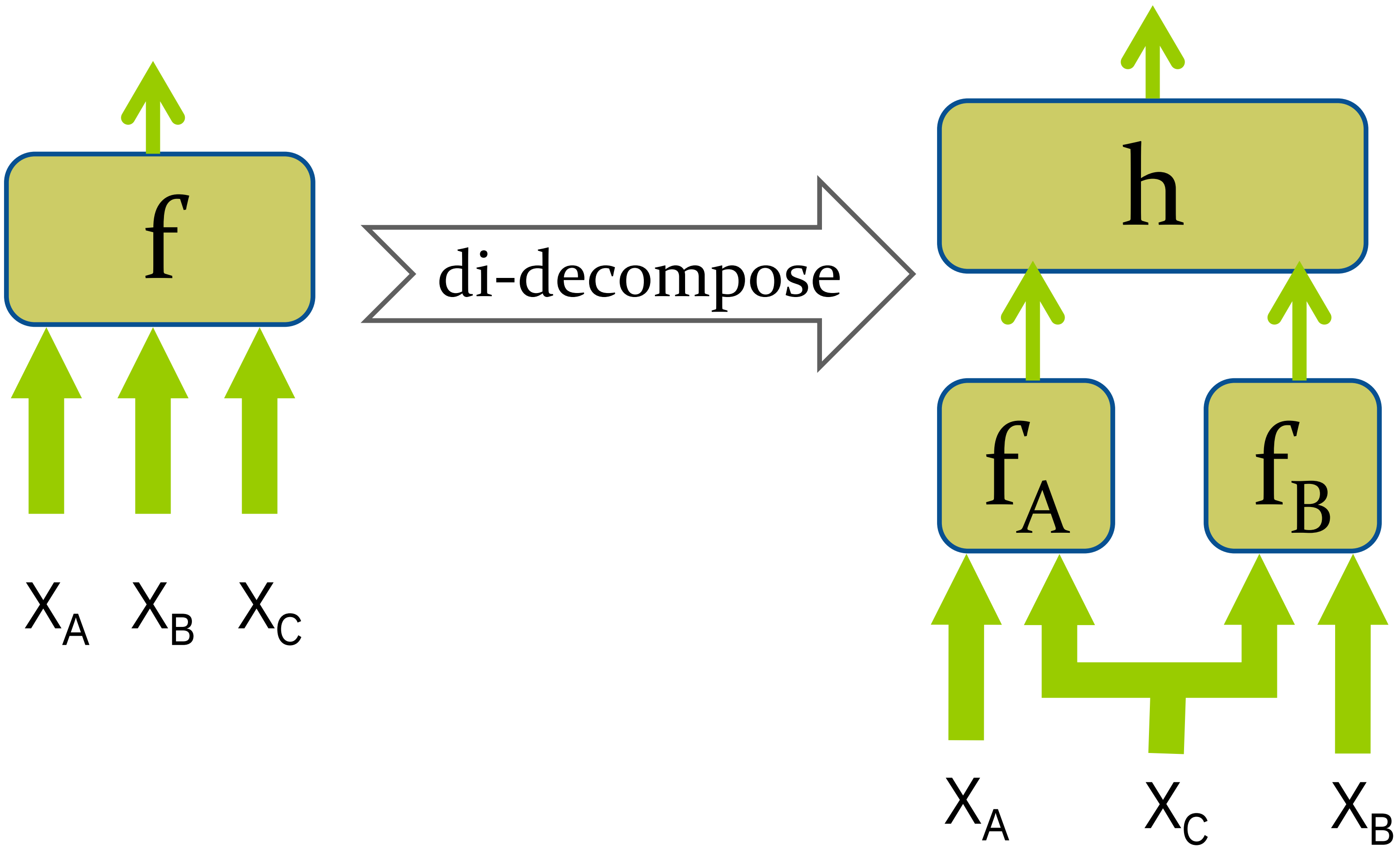


Summary

- Functional dependency is computable with pure SAT solving (with the help of Craig interpolation)
- Compared to BDD-based computation, it is much scalable to large designs

SAT & Logic Synthesis
Functional Bi-Decomposition

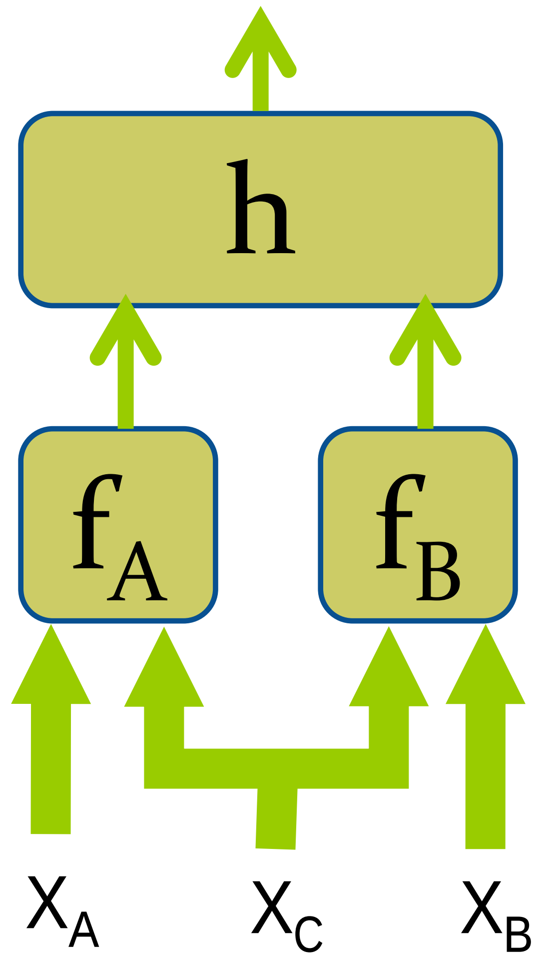
Bi-Decomposition



Bi-Decomposition

- A **variable partition** on $X = \{X_A|X_B|X_C\}$ has the property:
  - X_A , X_B , X_C are pair-wise disjoint, and
  - $X_A \cup X_B \cup X_C = X$

- If $X_C = \varnothing$, the decomposition is called **disjoint**; otherwise, **non-disjoint**



Bi-Decomposition

- We consider OR, AND, XOR bi-decompositions
  - These three cases are sufficient to generate any other type of bi-decomposition

a	**b**	**a+b**	**ab**	**$a \oplus b$**	**$a(\neg b)$**	**$a \oplus (\neg b)$**
0	0	0	0	0	0	1
0	1	1	0	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

Motivation

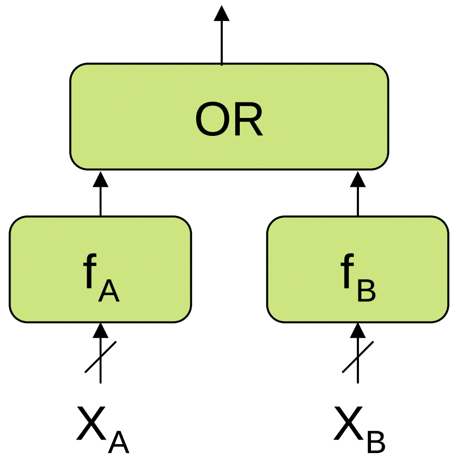
- Bi-decomposition breaks a large function into a network of smaller functions (necessary for FPGA implementation)
- Bi-decomposition can be applied to restructure logic network for optimization
  - It reduces circuit and communication complexity and thus simplify physical design

BDD-Based Computation

- Pros
  - Exact characterization of don't cares
- Cons
  - Memory explosion
  - Decomposability must be checked under a fixed variable partition

OR Bi-Decomposition

- Disjoint decomposition:
  $X_C = \varnothing$



- Example
  $f(a,b,c,d) = (\neg a)b+cd$

  $X = \{a,b,c,d\}=\{X_A|X_B\}$
  $X_A= \{a,b\}$, $X_B= \{c,d\}$

  $f(X) = (\neg a)b+cd$
  $= f_A(a,b)+f_B(c,d)$

$X_B \backslash X_A$	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	1	1	1	1
10	0	1	0	0

$f_B(X_B)$
0
0
1
0

$f_A(X_A)$	0	1	0	0

OR Bi-Decomposition

- f(X) can be written as $f_A(X_A) \vee f_B(X_B)$ if and only if, for every 1-entry in the decomposition table, 0-entries cannot appear simultaneously in the corresponding row and column

- Example
  $f(1101) = 0 = f_A(11) + f_B(01)$
  $f(0010) = 0 = f_A(00) + f_B(10)$
  $f(1110) = 1 = f_A(11) + f_B(10)$??

$X_B \backslash X_A$	00	01	11	10	$f_B(X_B)$
00	0	1	0	0	0
01	0	1	0	0	0
11	1	1	1	1	1
10	0	1	0	0	0
$f_A(X_A)$	0	1	0	0	

$X_B \backslash X_A$	00	01	11	10	$f_B(X_B)$
00	0	1	0	0	
01	0	1	0	0	
11	1	1	1	1	
10	0	1	1	0	?
$f_A(X_A)$			?		

SAT-Based OR Decomposition

- $\exists f_A, f_B$ such that $f(X) = f_A(X_A) \vee f_B(X_B)$
  $\Leftrightarrow$ For every 1-entry, no 0-entries can appear simultaneously in the corresponding row and column
  $\Leftrightarrow$ $f(X_A,X_B) \wedge \neg f(X_A',X_B) \wedge \neg f(X_A,X_B')$ is unsatisfiable

		X_A'		X_A			
	$X_B \backslash X_A$	00	01	11	10		$f_B(X_B)$
	00	0	1	0	0		?
X_B'	01	0	1	0	0		?
	11	1	1	1	1		?
X_B	10	0	1	1	0		?
	$f_A(X_A)$	?	?	?	?		

SAT-Based OR Decomposition

- $\exists f_A$, f_B such that $f(X) = f_A(X_A,X_C) \vee f_B(X_B,X_C)$
  $\Leftrightarrow$ Under every valuation of X_C, for every 1-entry, no 0-entries can appear simultaneously in the corresponding row and column
  $\Leftrightarrow$ $f(X_A,X_B,X_C) \wedge \neg f(X_A',X_B,X_C) \wedge \neg f(X_A,X_B',X_C)$ is unsatisfiable

X_C=00	X_A
X_B	

X_C=01	X_A
X_B	

X_C=10	X_A
X_B	

X_C=11	X_A
X_B	

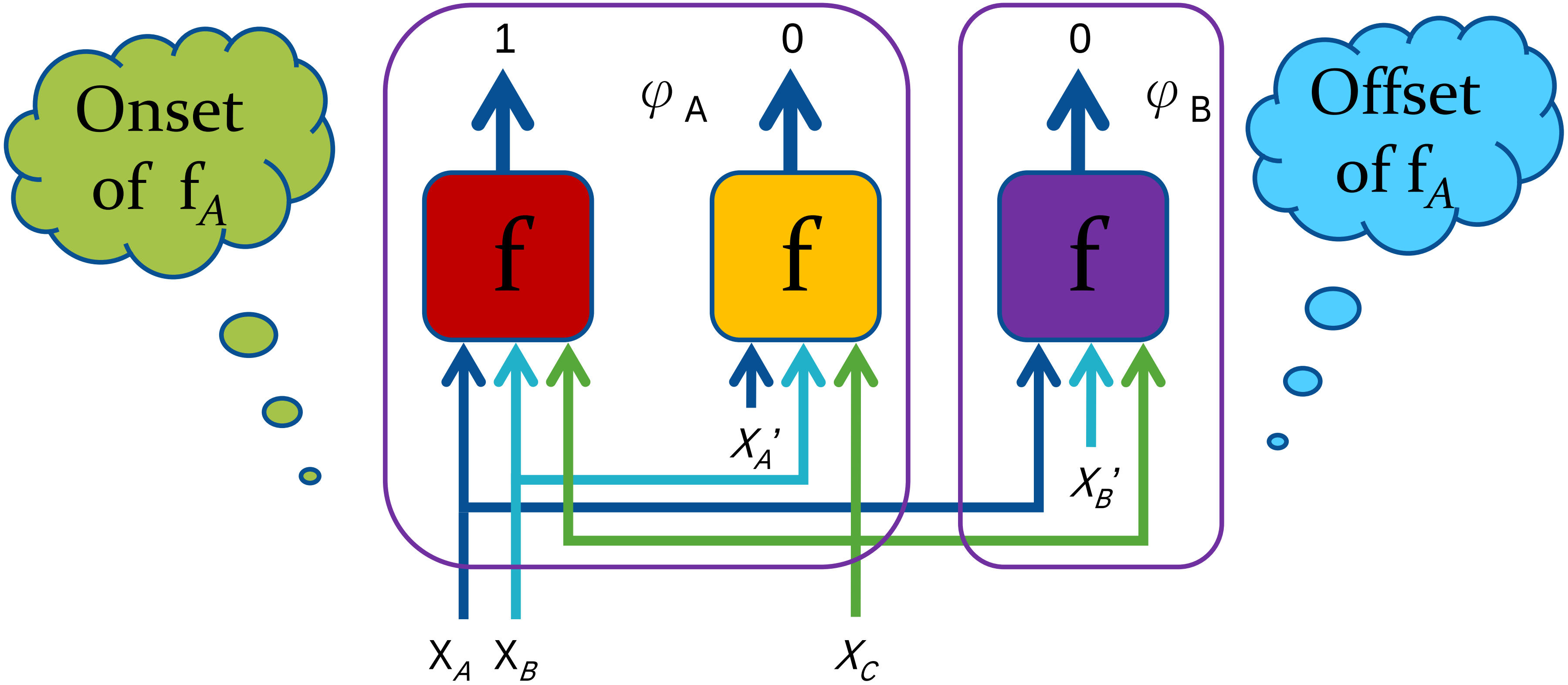
SAT-Based OR Decomposition

- $\exists f_A, f_B$ such that $f(X) = f_A(X_A, X_C) \vee f_B(X_B, X_C)$
  $\Leftrightarrow f(X_A, X_B, X_C) \wedge \neg f(X_A', X_B, X_C) \wedge \neg f(X_A, X_B', X_C)$ is UNSAT

- How to compute f_A and f_B? How to determine the variable partition?

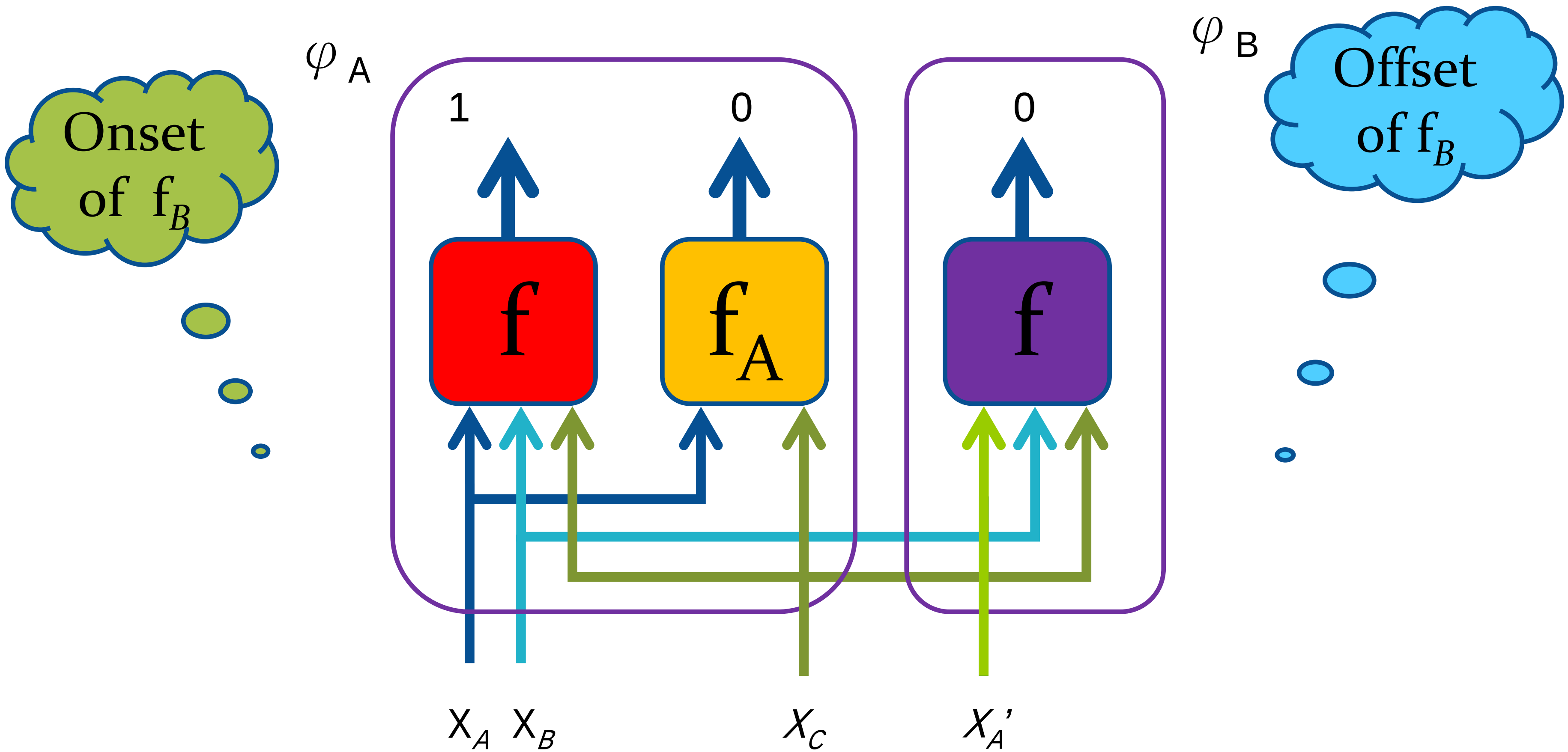
SAT-Based OR Decomposition
f_A Computation

- $f(X_A,X_B,X_C) \wedge \neg f(X_A',X_B,X_C) \wedge \neg f(X_A,X_B',X_C)$ is UNSAT



SAT-Based OR Decomposition
f_B Computation

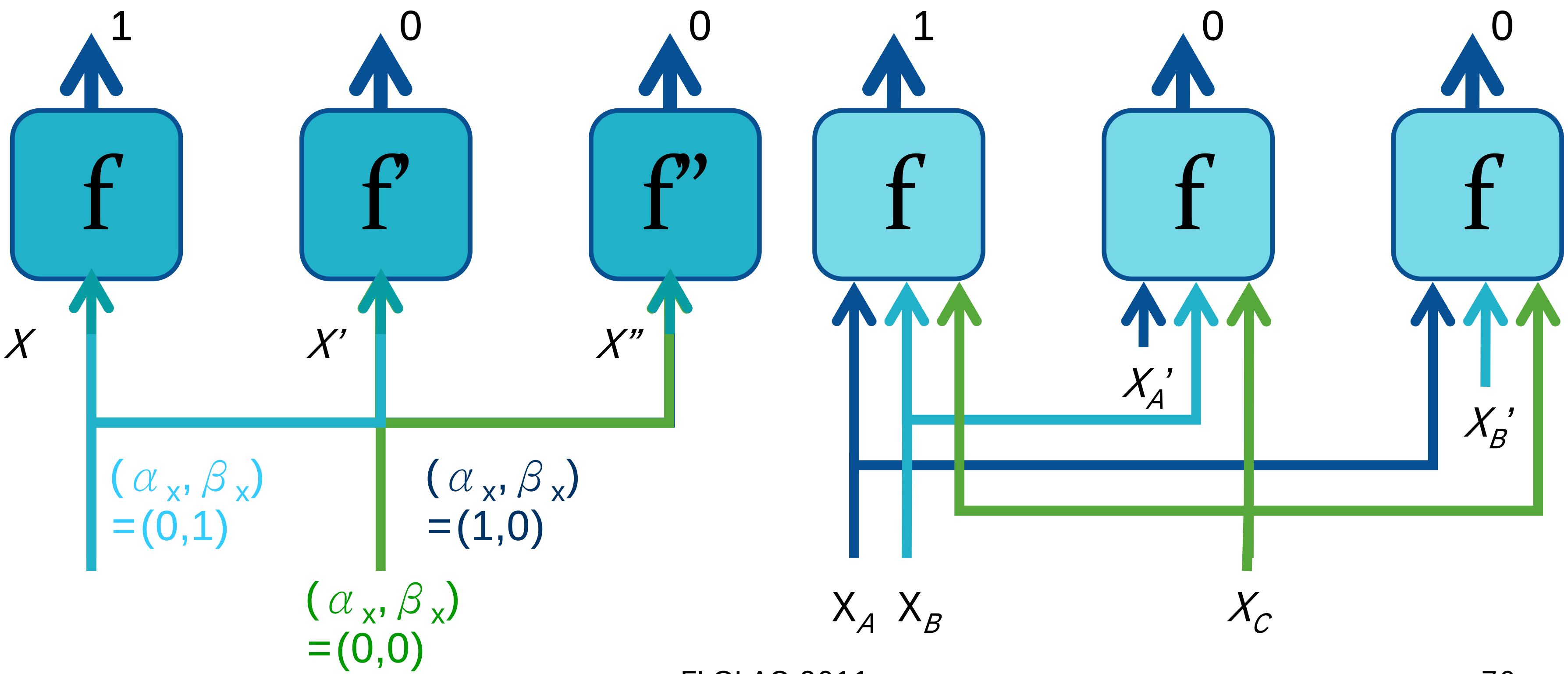
- $f(X_A,X_B,X_C) \wedge \neg f_A(X_A,X_C) \wedge \neg f(X_A',X_B,X_C)$ is UNSAT



SAT-Based OR Decomposition
Variable Partition

- $\varphi_A = f(X) \wedge \neg f(X') \wedge \bigwedge((x_i \equiv x'_i) \vee \alpha_{x_i})$
- $\varphi_B = \neg f(X'') \wedge \bigwedge((x_i \equiv x''_i) \vee \beta_{x_i})$

(α_x, β_x)	x is belongs to
(0,0)	X_C
(0,1)	X_B
(1,0)	X_A
(1,1)	either X_A or X_B



SAT-Based OR Decomposition
Variable Partition

- Make *unit assumption* on the control variables with MiniSat
  - Assume all the control variables are 0
  - SAT solver will return a conflict clause consisting of only the control variables
  - The conflict clause corresponds to a variable partition
- E.g.

  Conflict clause ($\alpha_{x_1}+\beta_{x_1}+\alpha_{x_2}+\beta_{x_3}$) indicates the unit assumption α_{x_1}=0, β_{x_1}=0, α_{x_2}=0,and β_{x_3}=0 causes unsatisfiability. So $x_1 \in X_C$, $x_2 \in X_B$, and $x_3 \in X_A$

SAT-Based OR Decomposition
Variable Partition

- Avoid trivial variable partition
  - Bi-decomposition trivially holds if X_C, $X_A \cup X_C$, or $X_B \cup X_C$ equals X
  - SAT solver may return a conflict clause that consists of all the control variables $\Rightarrow$ $X_C = X$
  - To avoid trivial partition, in unit assumption we specify two distinct variables x_a and x_b in X_A and X_B, respectively, and others in X_C initially
    - To check if a function is bi-decomposable, have to try at most C(n,2) iterations

SAT-Based AND Decomposition

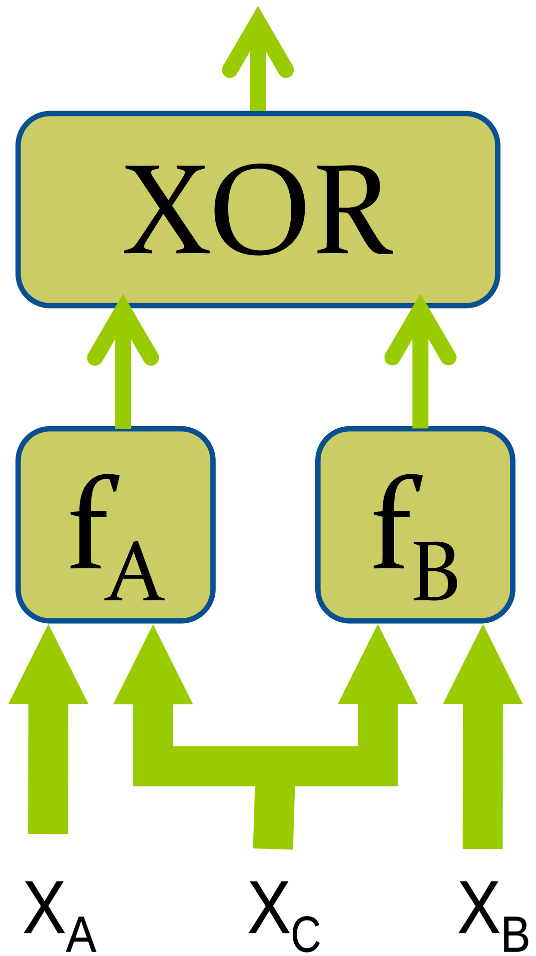
- $\exists f_A, f_B$ such that $f = f_A \wedge f_B$
  $\Leftrightarrow \exists f_A, f_B$ such that $\neg f = \neg f_A \vee \neg f_B$

- Example
  $f(a,b,c,d) = (a+\neg b+c)(b+\neg c+d)$
  $\neg f(a,b,c,d) = (\neg a)b(\neg c) \vee (\neg b)c(\neg d)$
  $= \neg f_A(a,b,c) \vee \neg f_B(b,c,d)$
  $f_A(a,b,c) = (a+\neg b+c)$, $f_B(b,c,d) = (b+\neg c+d)$
  $f(a,b,c,d) = f_A(a,b,c) \wedge f_B(b,c,d)$

SAT-Based XOR Decomposition

- (1)=(5)$\oplus$(7), (2)=(5)$\oplus$(8),(3)=(6)$\oplus$(7), (4)=(6)$\oplus$(8)
  $\Rightarrow$(1)$\oplus$(4)=(2)$\oplus$(3)
  $\Rightarrow$ (1)$\oplus$(2)=(3)$\oplus$(4)
  $\Rightarrow$ [(1)$\equiv$(2)]$\wedge$[(3)$\neq$(4)] UNSAT

		X_A'		X_A			
	X_B\X_A	00	01	11	10		$f_B(X_B)$
	00						
X_B'	01	(1)		(3)			(7)
	11						
X_B	10	(2)		(4)			(8)
	$f_A(X_A)$	(5)		(6)			



SAT-Based XOR Decomposition

- $[(1) \equiv (2)] \wedge [(3) \neq (4)]$ UNSAT
- $\exists f_A, f_B$ such that $f(X) = f_A(X_A,X_C) \oplus f_B(X_B,X_C) \Leftrightarrow$ $(f(X_A,X_B,X_C) \equiv f(X_A,X_B',X_C)) \wedge (f(X_A',X_B,X_C) \neq f(X_A',X_B',X_C))$ UNSAT

For every pair of columns (rows), their patterns are either complementary or identical to each other

		X_A,X_c		X_A',X_c	
	$X_B \backslash X_A$	00	01	11	10
	00				
X_B,X_c	01	(1)		(3)	
	11	$\equiv$		$\neq$	
X_B',X_c	10	(2)		(4)	

$X_B \backslash X_A$	00	01	11	10
00	1	0	1	1
01	0	1	0	0
11	0	1	0	0
10	1	0	1	1

SAT-Based XOR Decomposition
f_A, f_B Computation

- $f_A = f(X_A, 0, X_C)$
- $f_B = f(0, X_B, X_C) \oplus f(0, 0, X_C)$

X_c						
	$X_B \backslash X_A$	00	01	11	10	$f_B(X_B,X_c)$
	00	1	0	1	1	0
	01	0	1	0	0	1
	11	0	1	0	0	1
	10	1	0	1	1	0
	$f_A(X_A,X_c)$	1	0	1	1	

SAT-Based XOR Decomposition
Variable Partition

- Similar to OR decomposition
- $(f(X) \equiv f(X')) \wedge (f(X'') \neq f(X''')) \wedge$
  $(((x_i \equiv x_i'') \wedge (x_i' \equiv x_i''')) \vee \alpha_{x_i}) \wedge$
  $(((x_i \equiv x_i') \wedge (x_i'' \equiv x_i''')) \vee \beta_{x_i})$

$(\alpha_{x_i}, \beta_{x_i})$	X_i belongs to
(0,0)	X_C
(0,1)	X_B
(1,0)	X_A
(1,1)	either X_A or X_B

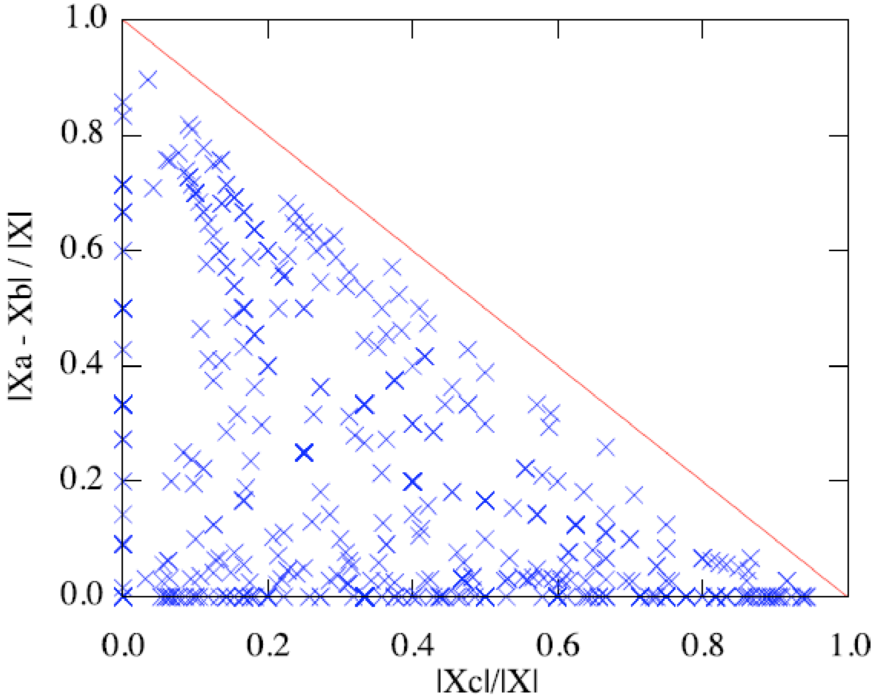
Practical Evaluation

				OR2-decomposition				XOR-decomposition			
circuit	#in	#max	#out	#dev	#slv	Time (sec)	Mem (Mb)	#dev	#slv	Time (sec)	Mem (Mb)
i2	201	201	1	1	1	1.07	18.6	1	34	2.16	18.59
s6669c	322	49	294	101	24423	198.14	29.13	176	3120	279.03	22.87
Dalu	75	75	16	1	26848	352.87	24.14	16	210	26.59	19.68
C880	60	45	26	16	222	8.36	20.72	11	4192	83.08	18.72

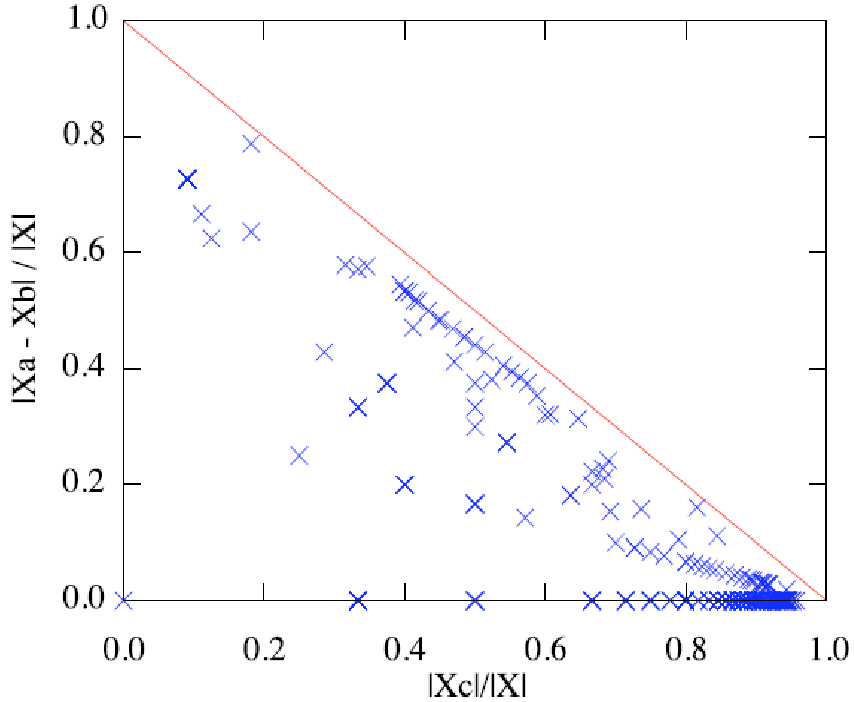
Practical Evaluation

Variable partition

OR decomposition



XOR decomposition



Summary

- OR, AND, XOR bi-decomposition can be formulated in terms of SAT solving
- Variable partitioning can be automated along the formulation
- SAT-based bi-decomposition is much more scalable than BDD-based methods

Quantified Satisfiability

Quantified Boolean Formula

- A quantified Boolean formula (QBF) is often written in **prenex form** (with quantifiers placed on the left) as

$$\underbrace{Q_1\, x_1, \ldots, Q_n\, x_n}_{\text{prefix}}.\ \underbrace{\varphi}_{\text{matrix}}$$

for $Q_i \in \{\forall, \exists\}$ and φ a quantifier-free formula

  - If φ is further in CNF, the corresponding QBF is in the so-called **prenex CNF** (PCNF), the most popular QBF representation
  - Any QBF can be converted to PCNF

Quantified Boolean Formula

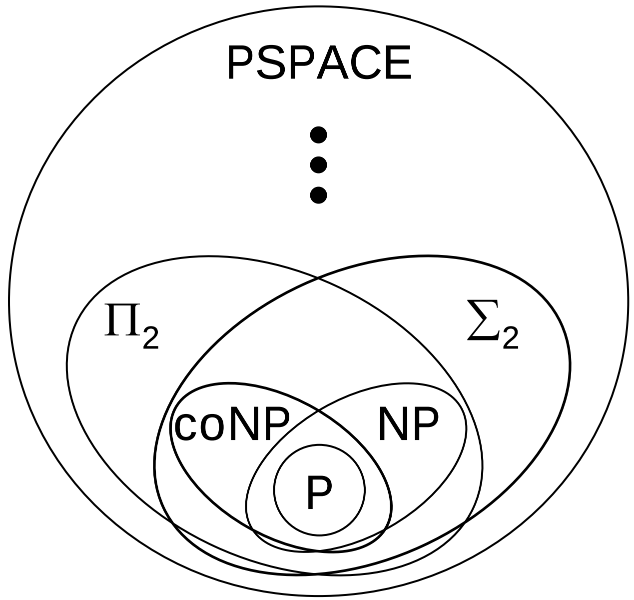
- Quantification order matters in a QBF
- A variable x_i in $(Q_1\ x_1, \ldots, Q_i\ x_i, \ldots, Q_n\ x_n.\ \varphi)$ is of **level** k if there are k quantifier alternations (i.e., changing from $\forall$ to $\exists$ or from $\exists$ to $\forall$) from Q_1 to Q_i.
  - Example

    $\forall a\ \exists b\ \forall c\ \forall d\ \exists e.\ \varphi$

    level(a)=0, level(b)=1, level(c)=2, level(d)=2, level(e)=3

Quantified Boolean Formula

- Many decision problems can be compactly encoded in QBFs
- In theory, QBF solving (QSAT) is PSPACE complete
  - The more the quantifier alternations, the higher the complexity in the Polynomial Hierarchy
- In practice, solvable QBFs are typically of size ~1,000 variables



QBF Solver

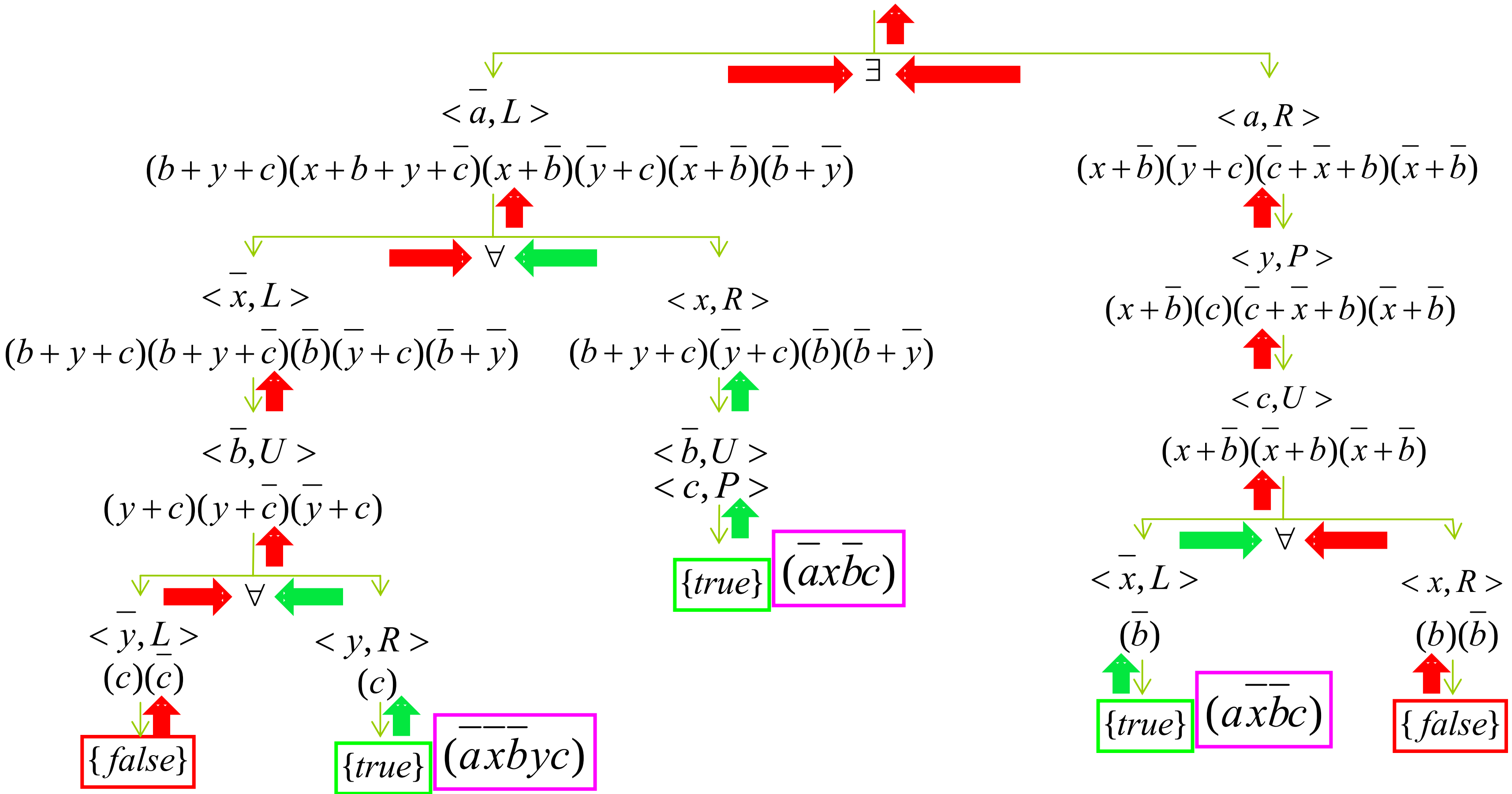
- QBF solver choices
  - Data structures for formula representation
    - **Prenex** vs. non-prenex
    - **Normal form** vs. non-normal form
      - CNF, NNF, BDD, AIG, etc.
  - Solving mechanisms
    - **Search**, Q-resolution, Skolemization, quantifier elimination, etc.
  - Preprocessing techniques

- Standard approach
  - Search-based PCNF formula solving (similar to SAT)
    - Both **clause learning** (from a conflicting assignment) and **cube learning** (from a satisfying assignment) are performed
      - Example
        $\forall a\, \exists b\, \exists c\, \forall d\, \exists e.\ (a+c)(\neg a+\neg c)(b+\neg c+e)(\neg b)(c+d+\neg e)(\neg c+e)(\neg d+e)$
        from 00101, we learn cube $\neg a \neg b c \neg d$ (can be further simplified to $\neg a$)

QBF Solving

- Example

$$\exists a \forall x \exists b \forall y \exists c \quad (a+b+y+c)(a+x+b+y+\bar{c})(x+\bar{b})(\bar{y}+c)(\bar{c}+\bar{a}+\bar{x}+b)(\bar{x}+\bar{b})(a+\bar{b}+\bar{y})$$



Q-Resolution

- **Q-resolution** on PCNF is similar to resolution on CNF, except that the pivots are restricted to existentially quantified variables and the additional rule of ∀**-reduction**

$$\frac{C_1 \vee x \qquad\qquad C_2 \vee \neg x}{\forall\text{-RED}(C_1 \vee C_2)}$$

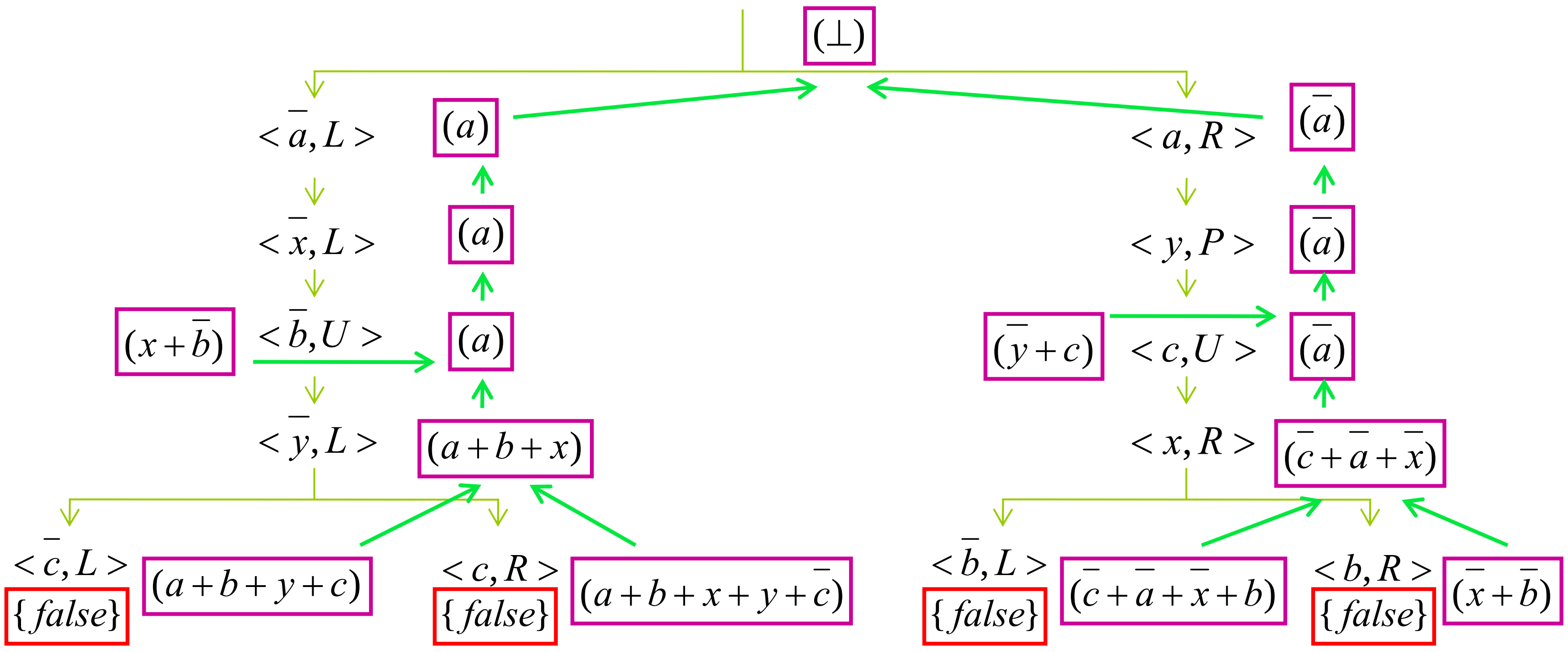
  where operator ∀-RED removes from $C_1 \vee C_2$ the universally (∀) quantified variables whose quantification levels are greater than any of the existentially (∃) quantified variables in $C_1 \vee C_2$
  - E.g.,
    prefix: $\forall a\ \exists b\ \forall c\ \forall d\ \exists e$
    $\forall\text{-RED}(a+b+c+d) = (a+b)$

- Q-resolution is complete for QBF solving
  - A PCNF formula is unsatisfiable if and only if there exists a Q-resolution sequence leading to the empty clause

Q-Resolution

- Example (cont'd)

$$\exists a \forall x \exists b \forall y \exists c \quad (a+b+y+c)(a+x+b+y+\bar{c})(x+\bar{b})(\bar{y}+c)(\bar{c}+\bar{a}+\bar{x}+b)(\bar{x}+\bar{b})(a+\bar{b}+\bar{y})$$



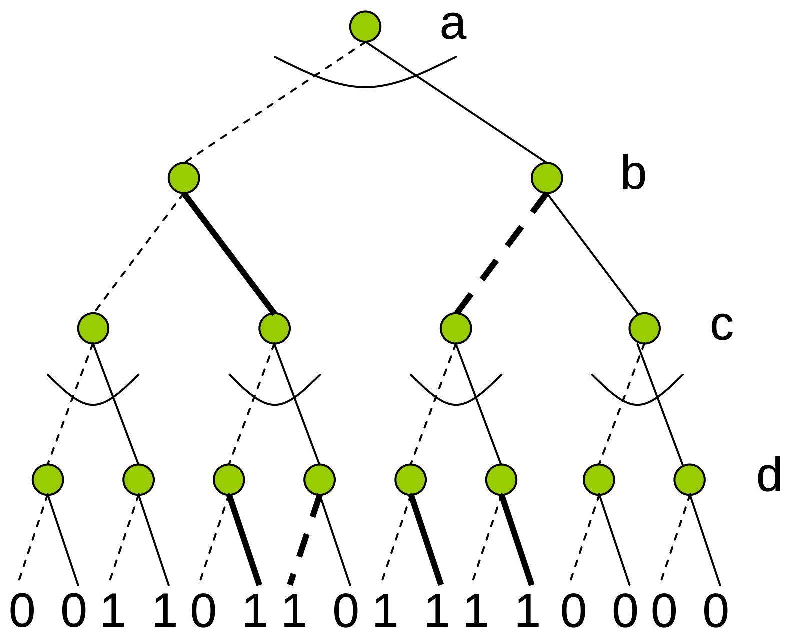
Skolemization

- Skolemization and Skolem normal form
  - Existentially quantified variables are replaced with function symbols
  - QBF prefix contains only two quantification levels
    - ∃ function symbols, ∀ variables
- Example

$\forall a\ \exists b\ \forall c\ \exists d.$
$(\neg a+\neg b)(\neg b+\neg c+\neg d)(\neg b+c+d)(a+b+c)$

Skolem functions

$\exists F_b(a)\ \exists F_d(a,c)\ \forall a\ \forall c.$
$(\neg a+\neg F_b)(\neg F_b+\neg c+\neg F_d)(\neg F_b+c+F_d)(a+F_b+c)$



QBF Certification

- QBF certification
  - Ensure correctness and, more importantly, provide useful information
  - Certificates
    - True QBF: term-resolution proof / Skolem-function (SF) model
      - SF model is more useful in practical applications
    - False QBF: clause-resolution proof / Herbrand-function (HF) countermodel
      - HF countermodel is more useful in practical applications
- Solvers and certificates
  - To date, only Skolemization-based solvers (e.g., sKizzo, squolem, Ebddres) can provide SFs
  - Search-based solvers (e.g., QuBE) are the most popular and can be instrumented to provide resolution proofs

QBF Certification

- Solvers and certificates

Solver	Algorithm	Certificate	
		True QBF	False QBF
QuBE-cert	search	Cube resolution	Clause resolution
yQuaffle	search	Cube resolution	Clause resolution
Ebddres	Skolemization	Skolem function	Clause resolution
sKizzo	Skolemization	Skolem function	-
squolem	Skolemization	Skolem function	Clause resolution

QBF Certification

- Incomplete picture of QBF certification

	Syntactic Certificate	Semantic Certificate
True QBF	Cube-resolution proof	Skolem-function model
False QBF	Clause-resolution proof	?

- Recent progress
  - Herbrand-function countermodel
    - [Balabanov, J, 2011 (ResQu)]
  - Syntactic to semantic certificate conversion
    - Linear time [Balabanov, J, 2011 (ResQu)]

QBF Certification

- Unified QBF certification

formula negation

True QBF $\longleftrightarrow$ **False QBF**

Cube resolution proof $\xrightarrow{\text{ResQu}}$ Skolem function (model)

Clause resolution proof $\xrightarrow{\text{ResQu}}$ Herbrand function (countermodel)

ResQu

- A Skolem-function model (Herbrand-function countermodel) for a true (false) QBF can be derived from its cube (clause) resolution proof

- A **Right-First-And-Or (RFAO) formula** is recursively defined as follows.

  φ := clause | cube | clause $\wedge$ φ | cube $\vee$ φ

  - E.g.,

    (a'+b) $\wedge$ ac $\vee$ (b'+c') $\wedge$ bc

    = ((a'+b) $\wedge$ (ac $\vee$ ((b'+c') $\wedge$ bc)))

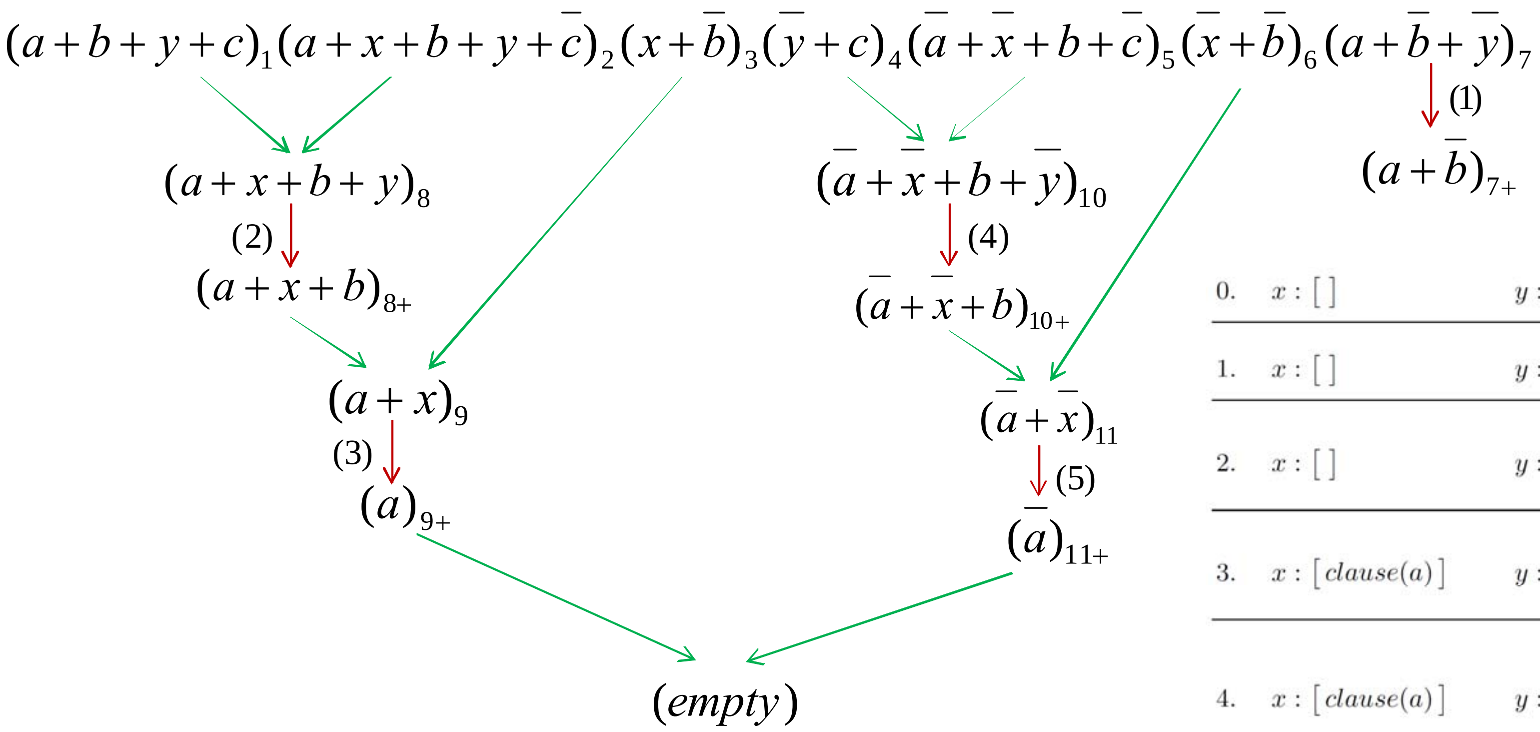
ResQu

Countermodel_construct

input: a false QBF Φ and its clause-resolution DAG $G_\Pi(V_\Pi, E_\Pi)$
output: a countermodel in RFAO formulas
begin
  foreach universal variable x of Φ
    `RFAO_node_array[`x`]` $:= \emptyset$;
  foreach vertex v of G_Π in topological order
    if $v.clause$ resulted from $\forall$-reduction on $u.clause$, i.e., $(u,v) \in E_\Pi$
      $v.cube := \neg(v.clause)$;
      foreach universal variable x reduced from $u.clause$ to get $v.clause$
        if x appears as positive literal in $u.clause$
          push $v.clause$ to `RFAO_node_array[`x`]`;
        else if x appears as negative literal in $u.clause$
          push $v.cube$ to `RFAO_node_array[`x`]`;
    if $v.clause$ is the empty clause
      foreach universal variable x of Φ
        simplify `RFAO_node_array[`x`]`;
      return `RFAO_node_array`'s;
end

ResQu

- Example
  - ∃a∀x∃b∀y∃c



0.	$x : [\,]$	$y : [\,]$
1.	$x : [\,]$	$y : [\,cube(\bar{a}b)\,]$
2.	$x : [\,]$	$y : \begin{bmatrix} cube(\bar{a}b), \\ clause(a+x+b) \end{bmatrix}$
3.	$x : [\,clause(a)\,]$	$y : \begin{bmatrix} cube(\bar{a}b), \\ clause(a+x+b) \end{bmatrix}$
4.	$x : [\,clause(a)\,]$	$y : \begin{bmatrix} cube(\bar{a}b), \\ clause(a+x+b), \\ cube(ax\bar{b}) \end{bmatrix}$
5.	$x : \begin{bmatrix} clause(a), \\ cube(a) \end{bmatrix}$	$y : \begin{bmatrix} cube(\bar{a}b), \\ clause(a+x+b), \\ cube(ax\bar{b}) \end{bmatrix}$

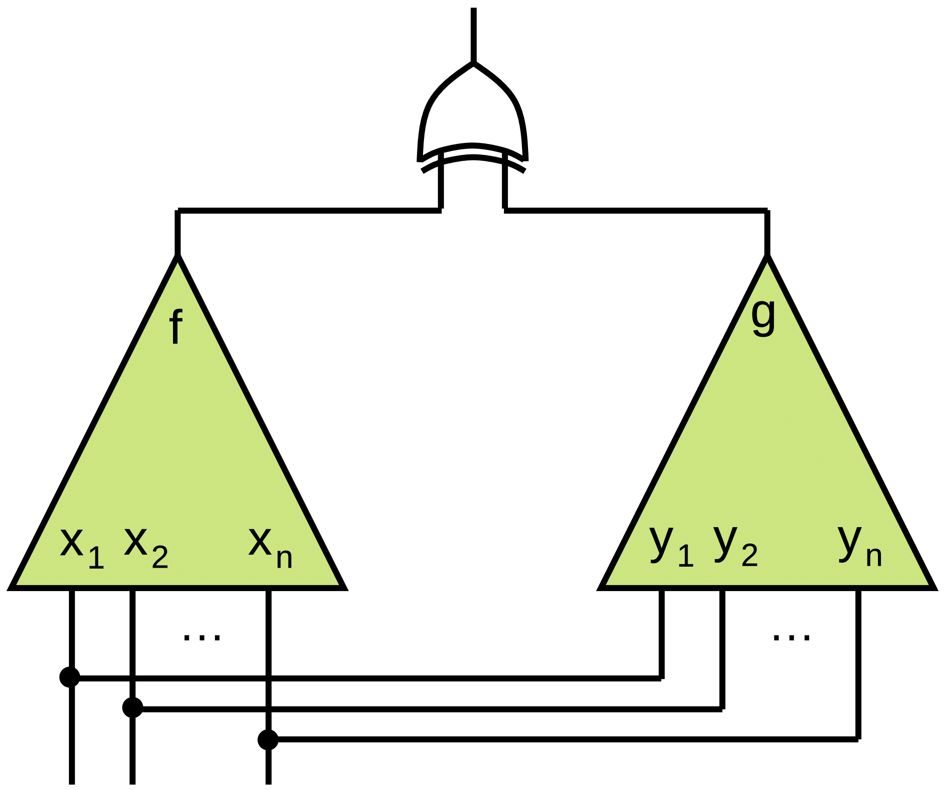
QBF Certification

- Applications of Skolem/Herbrand functions
  - Program synthesis
  - Winning strategy synthesis in two player games
  - Plan derivation in AI
  - Logic synthesis
  - …

QSAT & Logic Synthesis
Boolean Matching

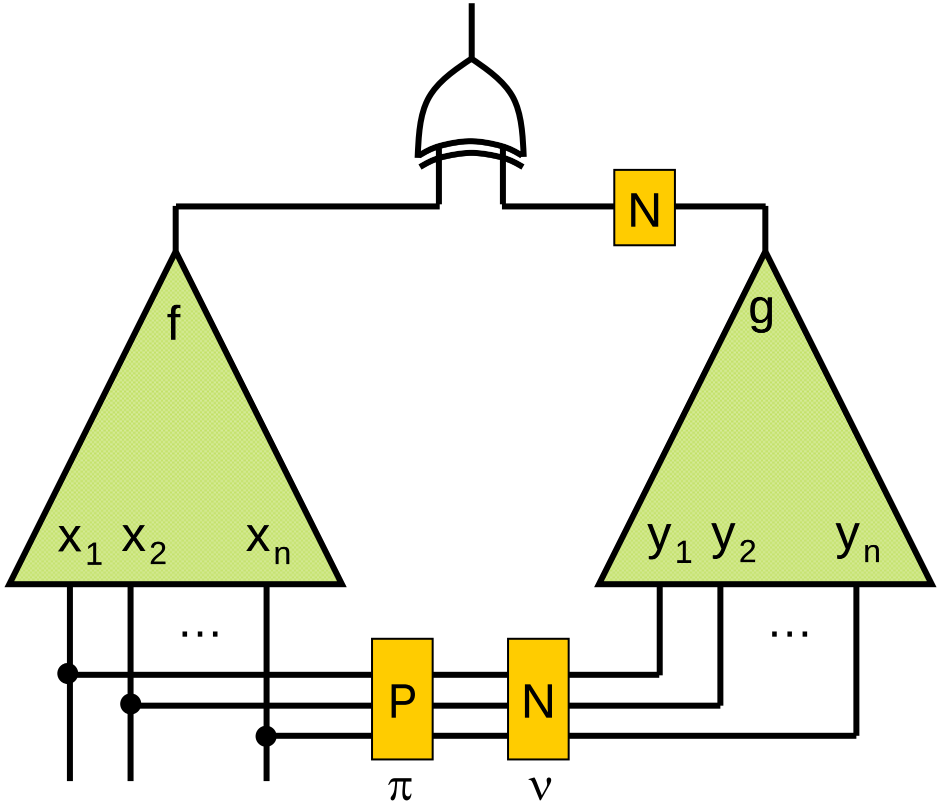
Introduction

- Combinational equivalence checking (CEC)
  - Known input correspondence
  - coNP-complete
  - Well solved in practical applications



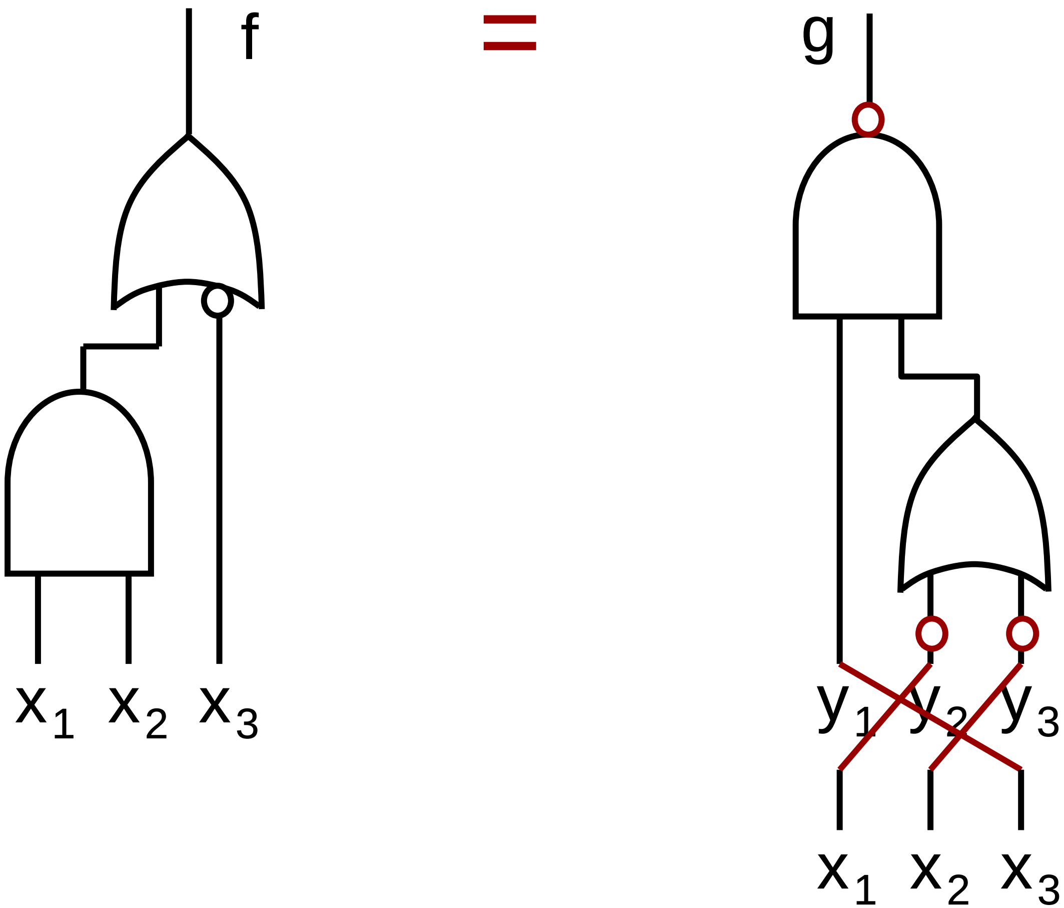
Introduction

- Boolean matching
  - P-equivalence
    - Unknown input permutation
    - O(n!) CEC iterations
  - NP-equivalence
    - Unknown input negation and permutation
    - $O(2^n n!)$ CEC iterations
  - NPN-equivalence
    - Unknown input negation, input permutation, and output negation
    - $O(2^{n+1} n!)$ CEC iterations



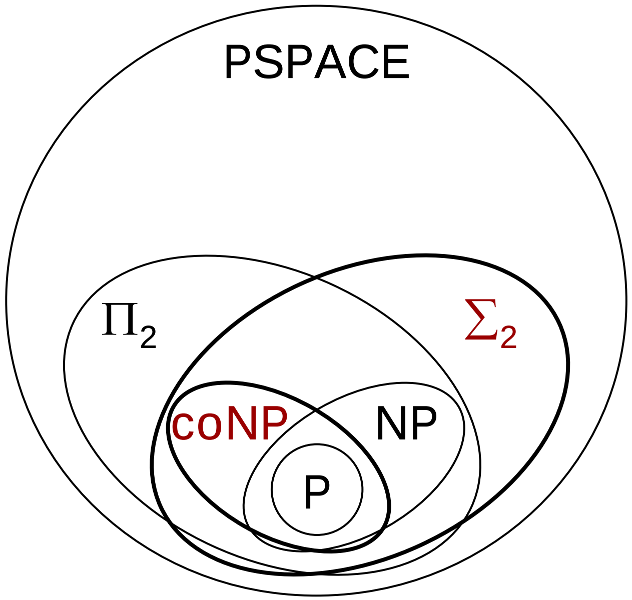
Introduction

- Example



Introduction

- Motivations
  - Theoretically
    - Complexity in between
      coNP (for all …) and
      Σ_2 (there exists … for all …)
      in the Polynomial Hierarchy (PH)
      - Special candidate to test PH collapse
    - Known as Boolean congruence/isomorphism dating back to the 19th century
  - Practically
    - Broad applications
      - Library binding
      - FPGA technology mapping
      - Detection of generalized symmetry
      - Logic verification
      - Design debugging/rectification
      - Functional engineering change order
    - Intensively studied over the last two decades



Introduction

Prior methods

	Complete ?	Function type	Equivalence type	Solution type	Scalability
Spectral methods	yes	CS	mostly P	one	– –
Signature based methods	no	mostly CS	P/NP	N/A	– ~ ++
Canonical-form based methods	yes	CS	mostly P	one	+
SAT based methods	yes	CS	mostly P	one/all	+
BooM (QBF/SAT-like)	yes	CS / IS	NPN	one/all	++

CS: completely specified
IS: incompletely specified

BooM: A Fast Boolean Matcher

- Features of BooM
  - General computation framework
  - Effective search space reduction techniques
    - **Dynamic learning** and **abstraction**
  - Theoretical SAT-iteration upper-bound:

$\cancel{O(2^n n!)}$ $O(2^{2n})$

Formulation

- Reduce NPN-equiv to 2 NP-equiv checks
  - Matching f and g; matching f and $\neg$g
- 2nd order formula of NP-equivalence

$$\exists \nu \circ \pi, \forall x \; ((f_c(x) \wedge g_c(\nu \circ \pi(x))) \Rightarrow (f(x) \equiv g(\nu \circ \pi(x))))$$

  - f_c and g_c are the care conditions of f and g, respectively
- Need 1st order formula instead for SAT solving

Formulation

- 0-1 matrix representation of $\nu \circ \pi$

$$
\begin{array}{c|cccccccc}
 & x_1 & \neg x_1 & x_2 & \neg x_2 & \cdots & x_n & \neg x_n \\
\hline
y_1 & a_{11} & b_{11} & a_{12} & b_{12} & \cdots & a_{1n} & b_{1n} \\
y_2 & a_{21} & b_{21} & a_{22} & b_{22} & \cdots & a_{2n} & b_{2n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
y_n & a_{n1} & b_{n1} & a_{n2} & b_{n2} & \cdots & a_{nn} & b_{nn}
\end{array}
$$

$\sum = 1$

$\sum = 1$

$a_{ij} \Rightarrow (x_j \equiv y_i)$

$b_{ij} \Rightarrow (\neg x_j \equiv y_i)$

Formulation

- Quantified Boolean formula (QBF) for NP-equivalence

  $$\exists a, \exists b, \forall x, \forall y\ (\varphi_C \wedge \varphi_A \wedge ((f_c \wedge g_c) \Rightarrow (f \equiv g))$$

  - φ_C: cardinality constraint
  - φ_A: $/\backslash_{i,j}\ (a_{ij} \Rightarrow (y_i \equiv x_j))\ (b_{ij} \Rightarrow (y_i \equiv \neg x_j))$

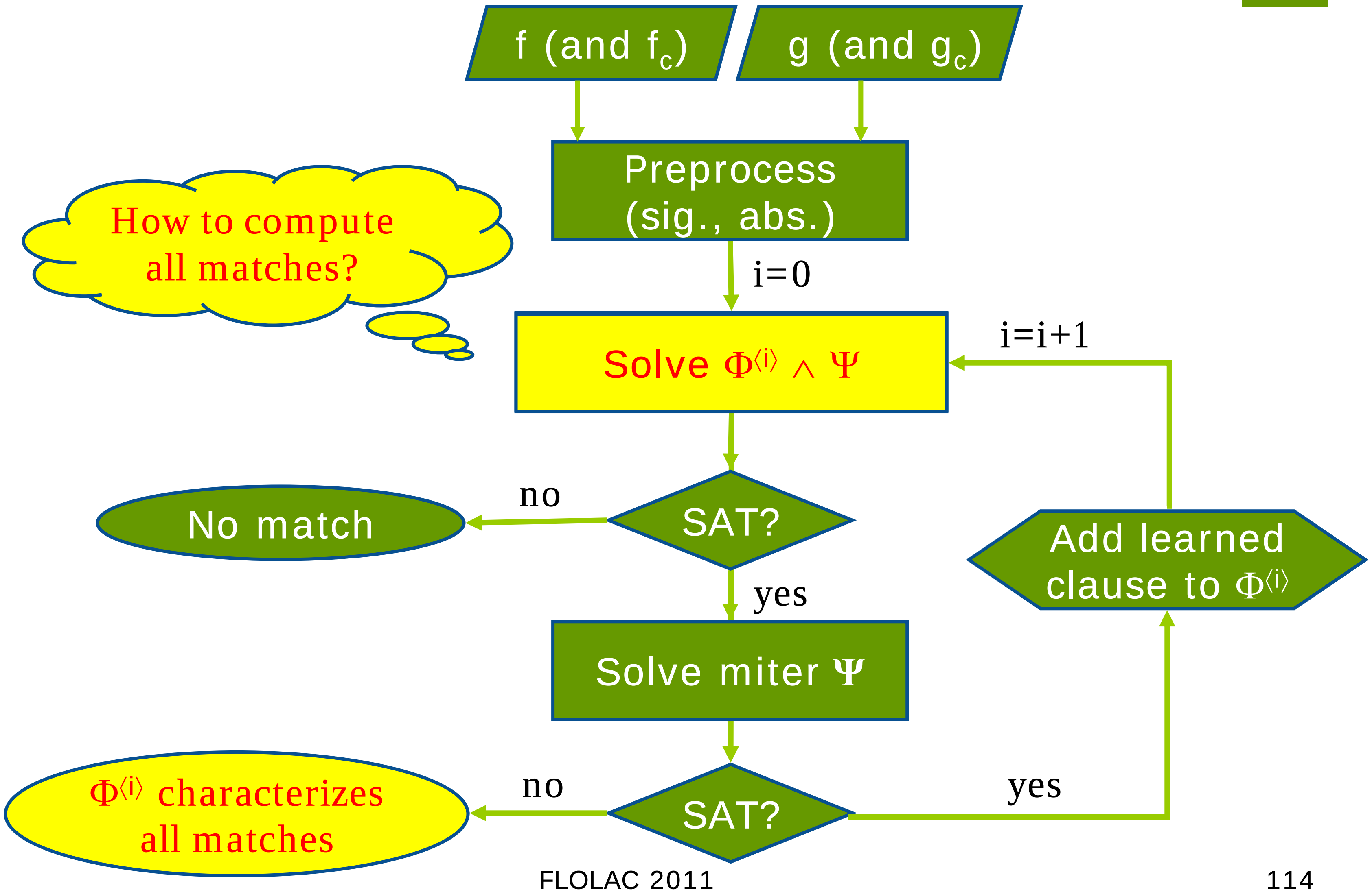
- Look for an assignment to a- and b-variables that satisfies φ_C and makes the **miter constraint**

  $$\Psi = \varphi_A \wedge (f \neq g) \wedge f_c \wedge g_c$$

  unsatisfiable

- Refine φ_C iteratively in a sequence $\Phi^{\langle 0\rangle}, \Phi^{\langle 1\rangle}, \ldots, \Phi^{\langle k\rangle}$, for $\Phi^{\langle i+1\rangle} \Rightarrow \Phi^{\langle i\rangle}$ through **conflict-based learning**

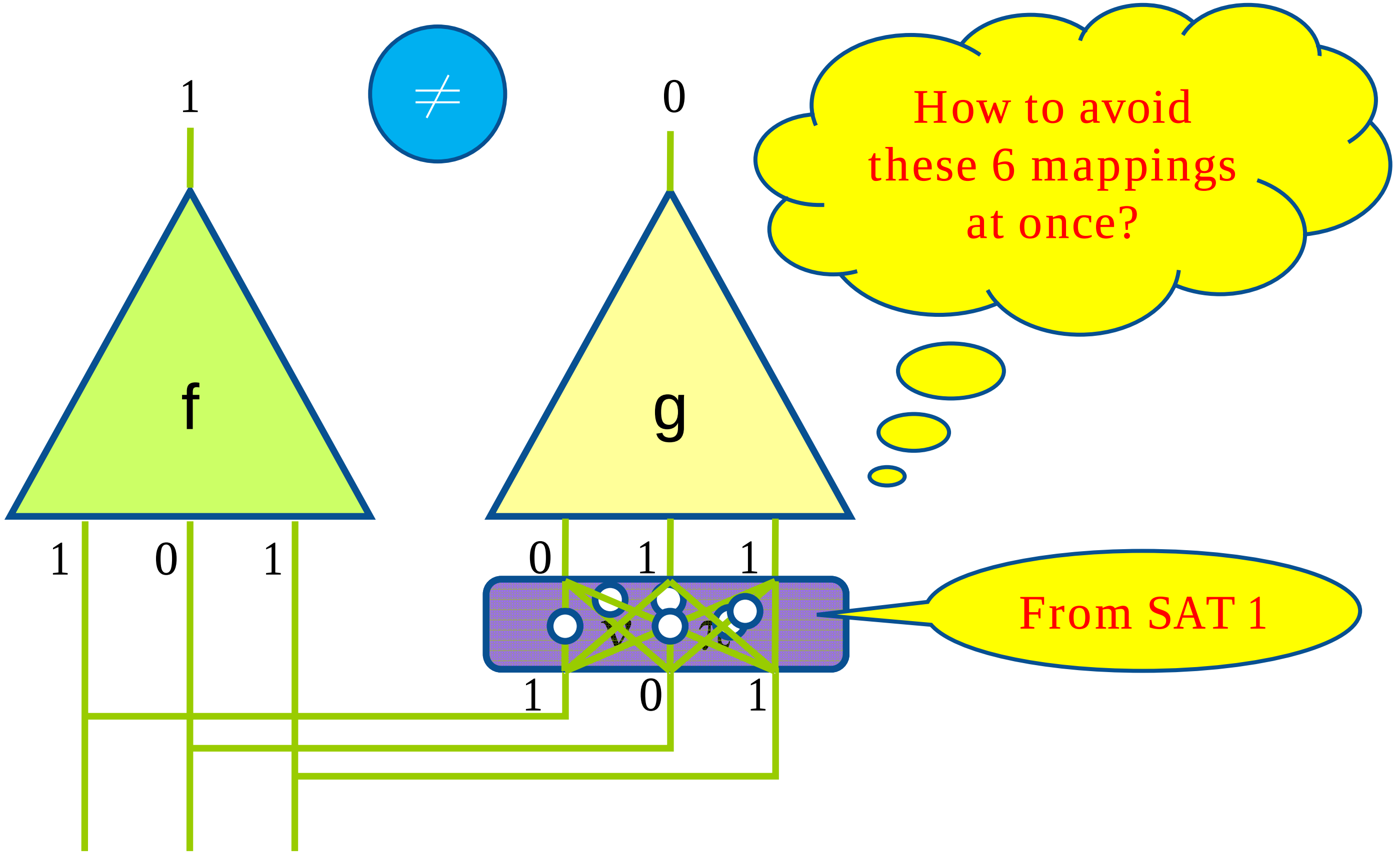
BooM Flow



f (and f$_c$)
g (and g$_c$)
Preprocess (sig., abs.)
How to compute all matches?
i=0
Solve $\Phi^{\langle i \rangle} \wedge \Psi$
i=i+1
no
No match
SAT?
yes
Add learned clause to $\Phi^{\langle i \rangle}$
Solve miter Ψ
no
$\Phi^{\langle i \rangle}$ characterizes all matches
SAT?
yes

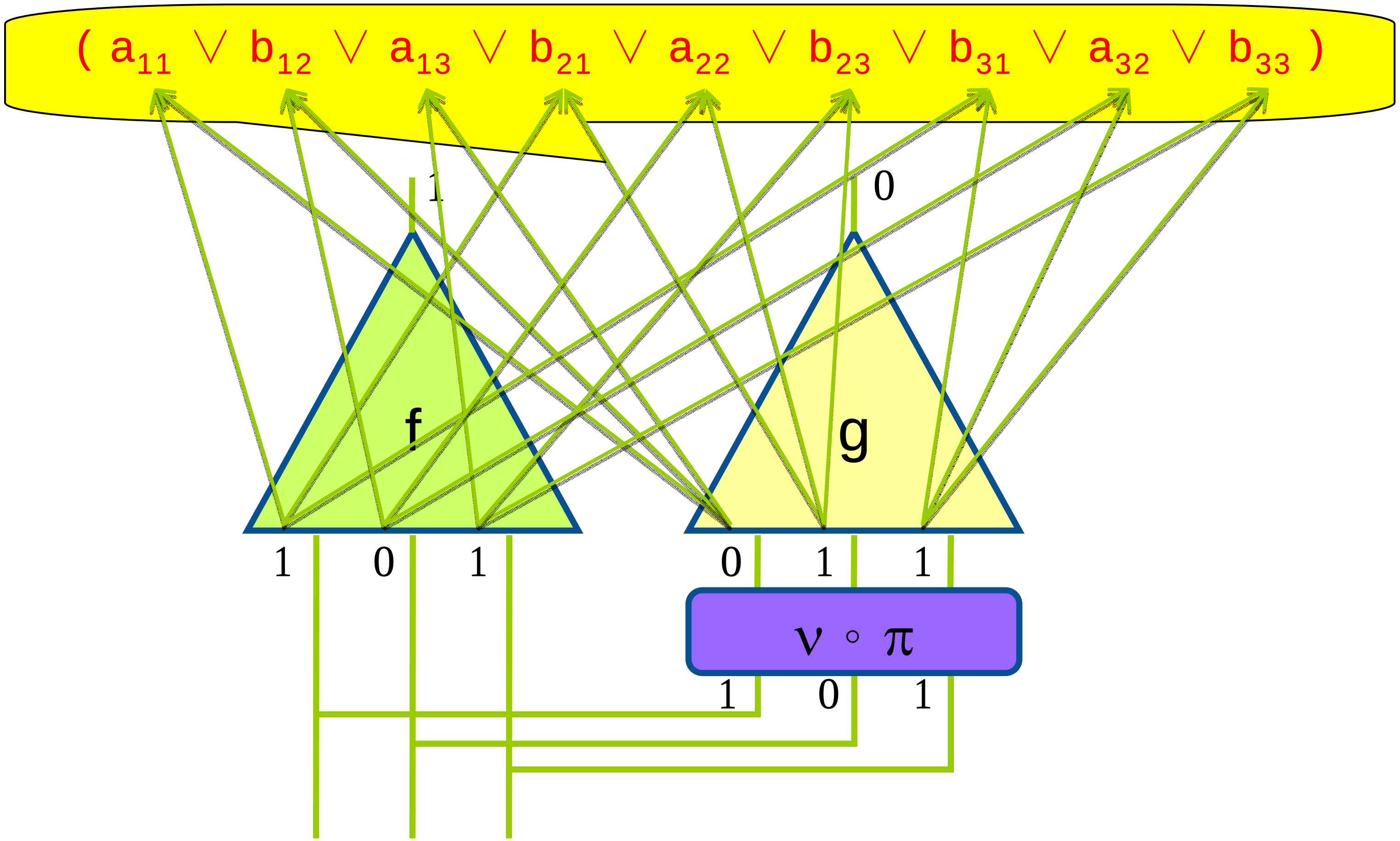
NP-Equivalence
Conflict-based Learning

- Observation



NP-Equivalence
Conflict-based Learning

- Learnt clause generation



NP-Equivalence
Conflict-based Learning

- Proposition:

  If $f(u) \neq g(v)$ with $v = \nu \circ \pi(u)$ for some $\nu \circ \pi$ satisfying $\Phi^{\langle i \rangle}$, then the learned clause $\bigvee_{ij} l_{ij}$ for literals

  $l_{ij} = (v_i \neq u_j)\ ?\ a_{ij} : b_{ij}$

  excludes from $\Phi^{\langle i \rangle}$ the mappings $\{\nu' \circ \pi' \mid \nu' \circ \pi'(u) = \nu \circ \pi(u)\}$

- Proposition:

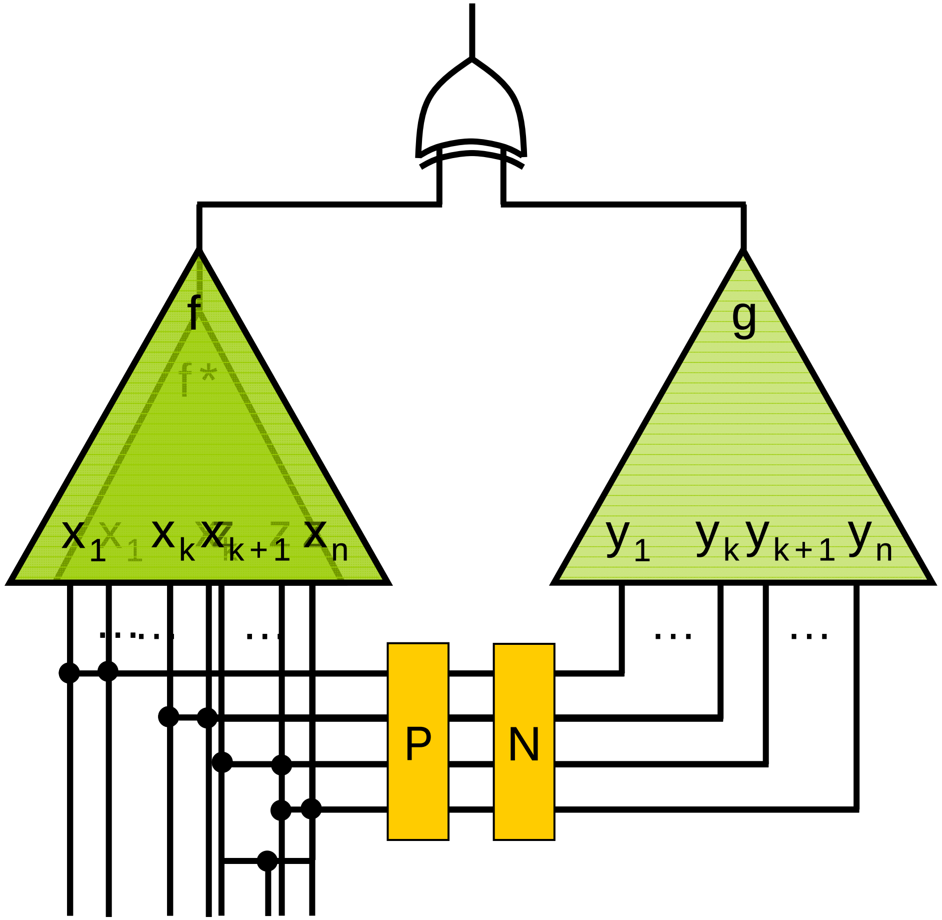
  The learned clause prunes n! infeasible mappings

- Proposition:

  The refinement process $\Phi^{\langle 0 \rangle}, \Phi^{\langle 1 \rangle}, \ldots, \Phi^{\langle k \rangle}$ is bounded by 2^{2n} iterations

NP-Equivalence Abstraction

- Abstract Boolean matching
  - Abstract $f(x_1,\ldots,x_k,x_{k+1},\ldots,x_n)$ to $f(x_1,\ldots,x_k,z,\ldots,z) = f^*(x_1,\ldots,x_k,z)$
  - Match $g(y_1,\ldots,y_n)$ against $f^*(x_1,\ldots,x_k,z)$
  - Infeasible matching solutions of f* and g are also infeasible for f and g



NP-Equivalence
Abstraction

- Abstract Boolean matching
  - Similar matrix representation of negation/permutation

$$
\begin{array}{c@{}c}
 & \begin{array}{cccccccc} x_1^* & \neg x_1^* & \cdots & x_k^* & \neg x_k^* & z & \neg z \end{array} \\
\begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array} &
\left(\begin{array}{cc|c|cc|cc}
a_{11} & b_{11} & \cdots & a_{1k} & b_{1k} & a_{1(k+1)} & b_{1(k+1)} \\
a_{21} & b_{21} & \cdots & a_{2k} & b_{2k} & a_{2(k+1)} & b_{2(k+1)} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\
a_{n1} & b_{n1} & \cdots & a_{nk} & b_{nk} & a_{n(k+1)} & b_{n(k+1)}
\end{array}\right)
\end{array}
\quad \sum = 1
$$

$\sum = 1$

- Similar cardinality constraints, except for allowing multiple y-variables mapped to z

NP-Equivalence Abstraction

- Used for preprocessing
- Information learned for abstract model is valid for concrete model
- Simplified matching in reduced Boolean space

P-Equivalence
Conflict-based Learning

- Proposition:

  If $f(u) \neq g(v)$ with $v = \pi(u)$ for some π satisfying $\Phi^{\langle i \rangle}$, then the learned clause $\bigvee_{ij} l_{ij}$ for literals

  $l_{ij} = (v_i\!=\!0 \text{ and } u_j\!=\!1)\ ?\ a_{ij} : \emptyset$

  excludes from $\Phi^{\langle i \rangle}$ the mappings $\{\pi' \mid \pi'(u) = \pi(u)\}$

P-Equivalence Abstraction

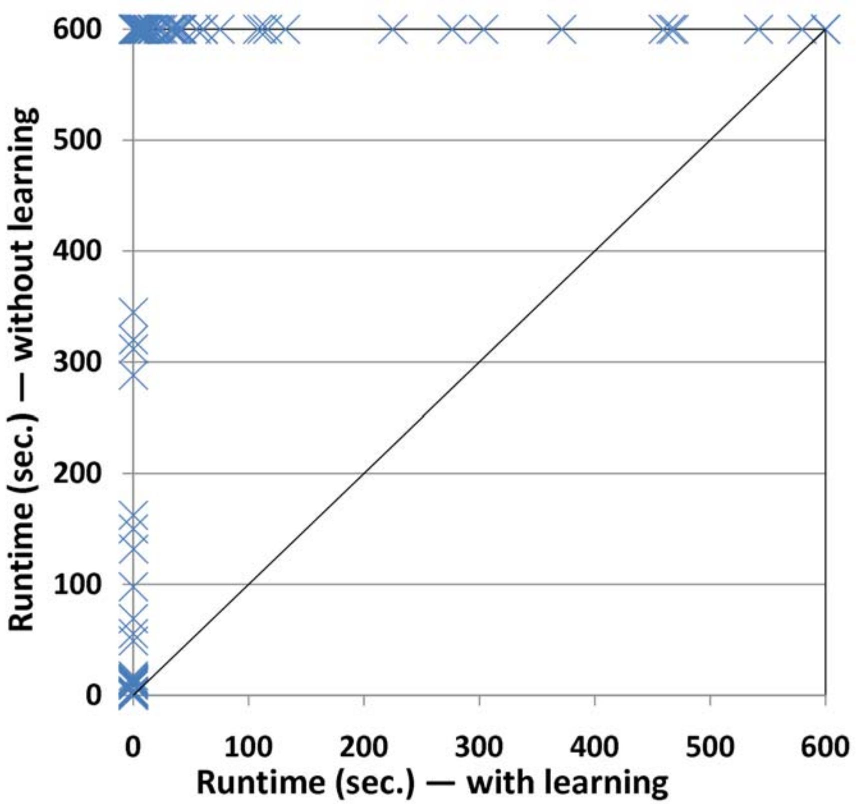
- Abstraction enforces search in biased truth assignments and makes learning strong
  - For f* having k support variables, a learned clause converted back to the concrete model consists of at most (k–1)(n–k+1) literals

Practical Evaluation

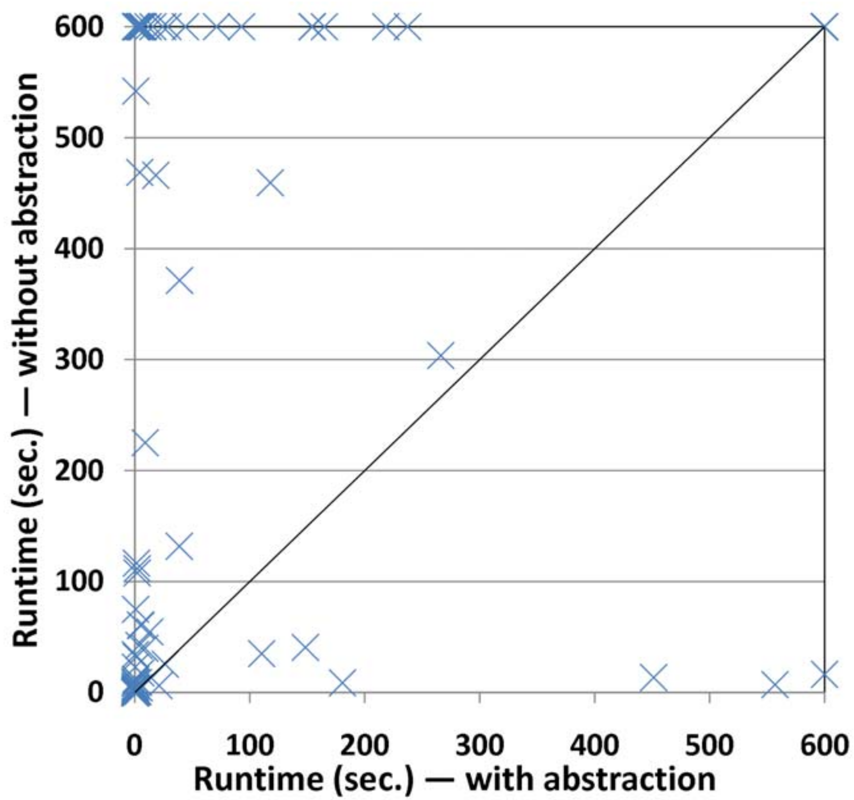
- BooM implemented in ABC using MiniSAT
- A function is matched against its synthesized, and input-permuted/negated version
  - Match individual output functions of MCNC, ISCAS, ITC benchmark circuits
    - 717 functions with 10~39 support variables and 15~2160 AIG nodes
  - Time-limit 600 seconds
  - Baseline preprocessing exploits symmetry, unateness, and simulation for initial matching

Practical Evaluation

Learning



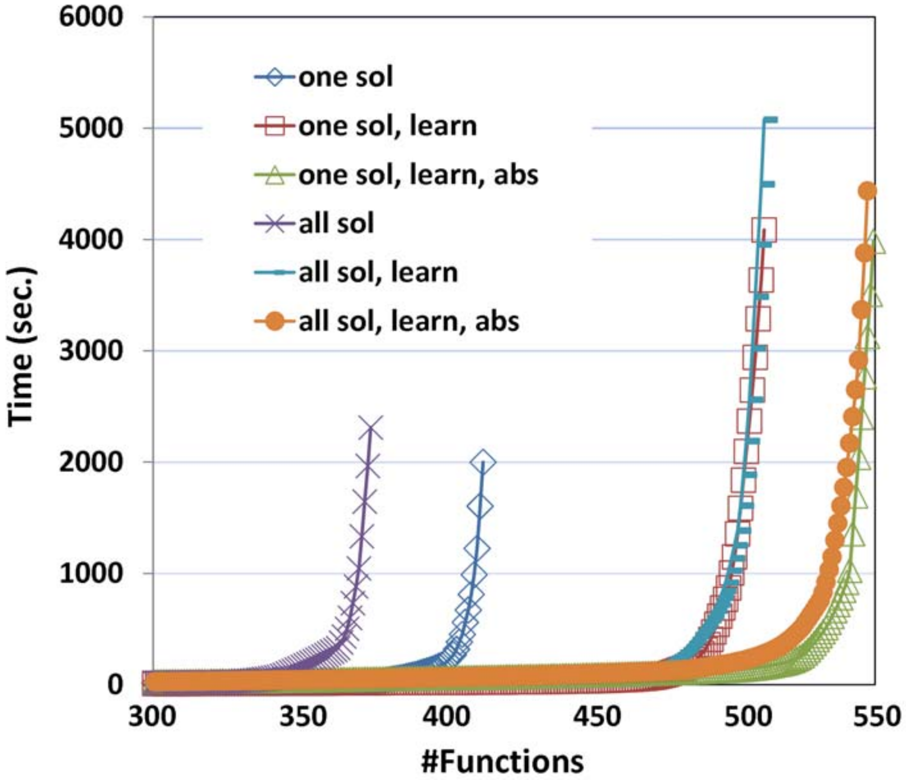
Abstraction



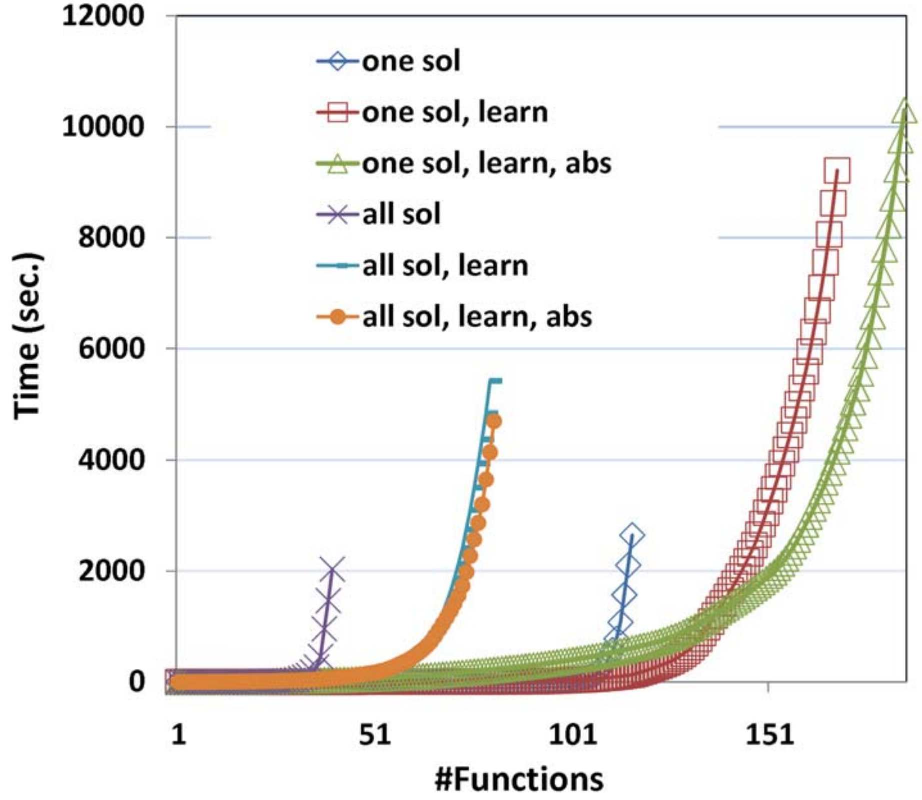
(P-equivalence; find all matches)

Practical Evaluation

P-equivalence

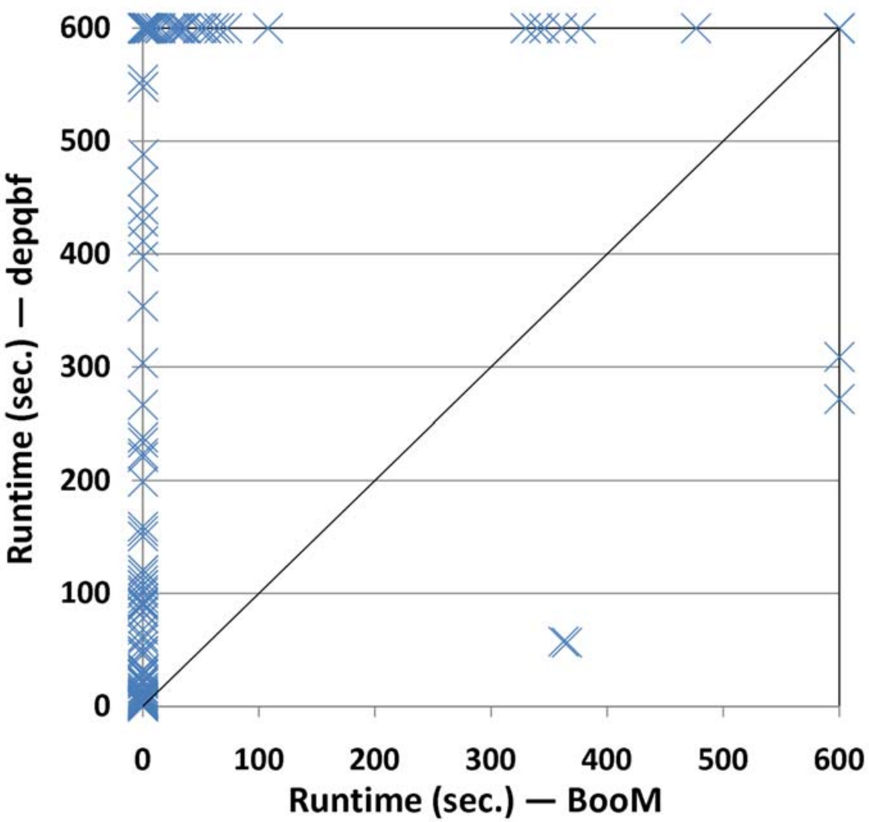


NP-equivalence



Practical Evaluation

BooM vs. DepQBF



(runtime after same preprocessing; P-equivalence; find one match)

Conclusions

- BooM, a dedicated decision procedure for Boolean matching
  - Effective learning and abstraction
    - Far faster than state-of-the-art QBF solver
    - Theoretical upper bound reduced from $O(2^n n!)$ to $O(2^{2n})$
      - Empirically exponent ~7 times less for P, ~3 times less for NP
  - General computation framework
    - Handles NPN-equivalence, incompletely specified functions
    - Allows easy integration with signature based methods

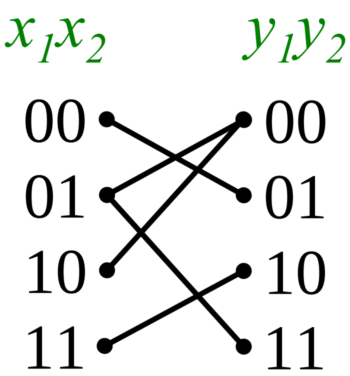
- Anticipate BooM to be a common platform for other Boolean matching developments and to facilitate practical applications

QSAT & Logic Synthesis

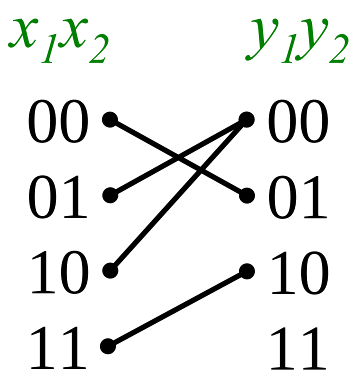
Relation Determinization

Relation vs. Function

- Relation *R(X, Y)*
  - Allow one-to-many mappings
    - Can describe non-deterministic behavior
  - More generic than functions



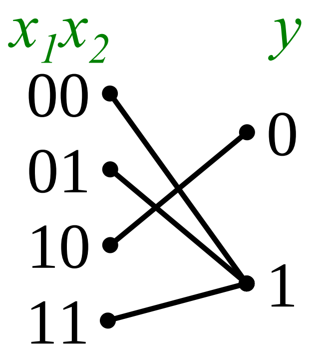
- Function *F(X)*
  - Disallow one-to-many mappings
    - Can only describe deterministic behavior
  - A special case of relation



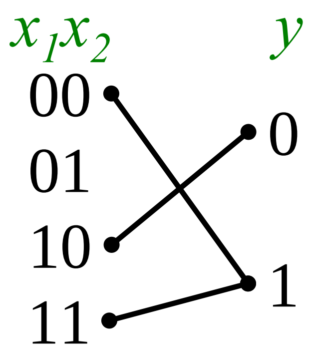
$f_1 = x_1\, x_2$
$f_2 = \neg x_1 \neg x_2$

Relation

- Total relation
  - Every input element is mapped to at least one output element

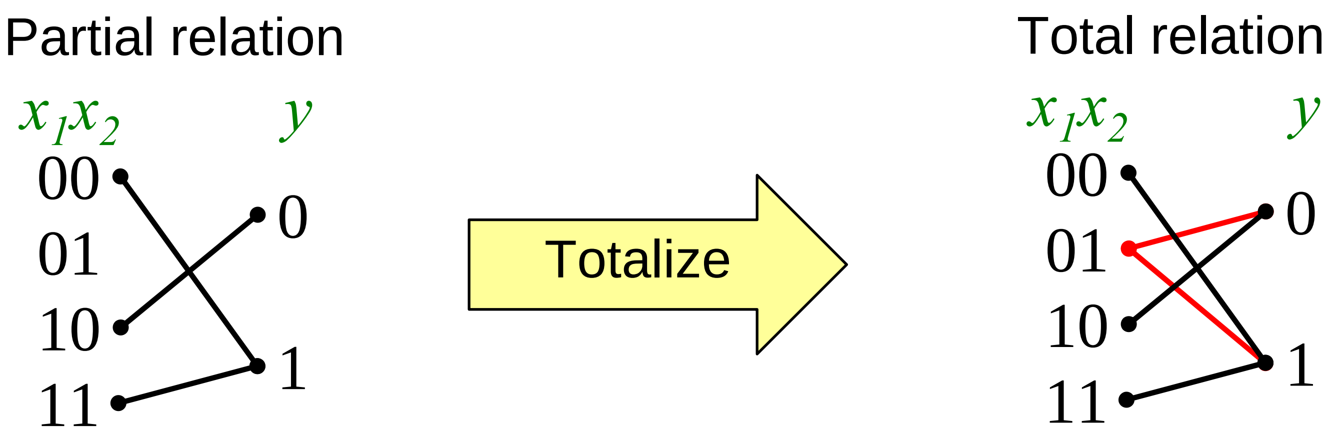


- Partial relation
  - Some input element is not mapped to any output element



Relation

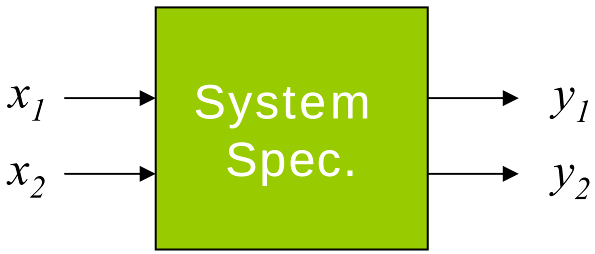
- A partial relation can be **totalized**
  - Assume that the input element not mapped to any output element is a don't care

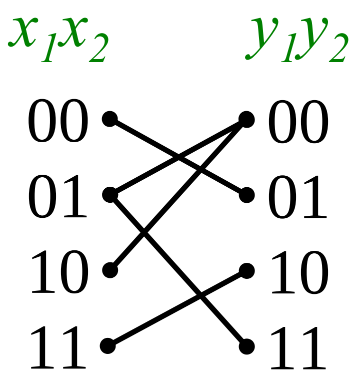


$$T(X, y) = R(X, y) \vee \forall y.\ \neg R(X, y)$$

Motivation

- Applications of Boolean relation
  - In high-level design, Boolean relations can be used to describe (nondeterministic) specifications
  - In gate-level design, Boolean relations can be used to characterize the flexibility of sub-circuits
    - Boolean relations are more powerful than traditional don't-care representations



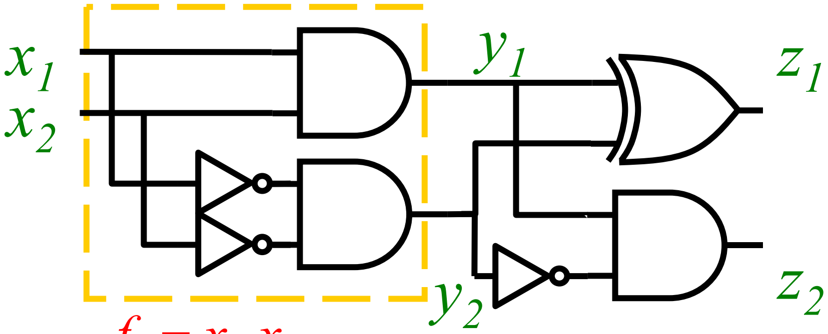


Motivation

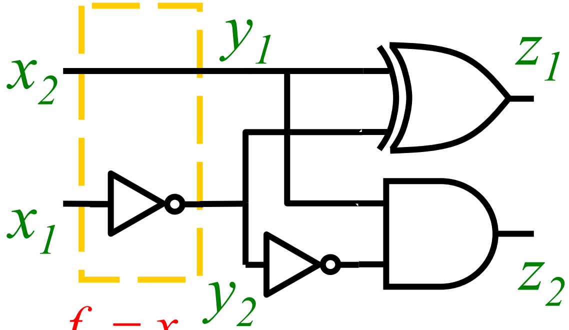
- Relation determinization
  - For hardware implement of a system, we need functions rather than relations
    - Physical realization are deterministic by nature
    - One input stimulus results in one output response
  - To simplify implementation, we can explore the flexibilities described by a relation for optimization

Motivation

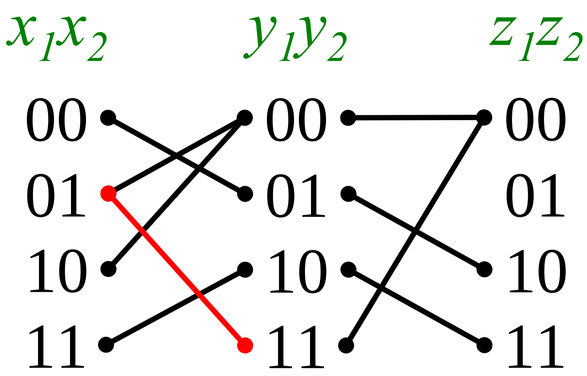
- Example



$f_1 = x_1\, x_2$
$f_2 = \neg x_1 \neg x_2$



$f_1 = x_2$
$f_2 = \neg x_1$



Relation Determinization

- Given a *nondeterministic* Boolean relation $R(X, Y)$, how to determinize and extract functions from it?

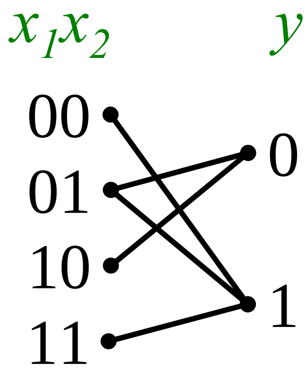
- For a deterministic total relation, we can uniquely extract the corresponding functions

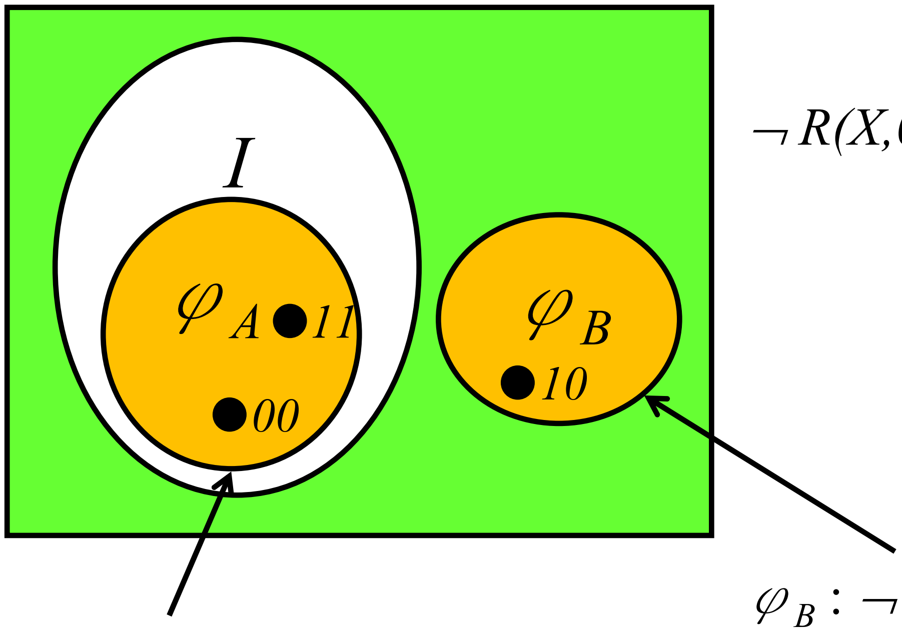
Relation Determinization

- Approaches to relation determinization
  - Iterative method (determinize one output at a time)
    - BDD- or SOP-based representation
      - Not scalable
      - Better optimization
    - AIG representation
      - Focus on scalability with reasonable optimization quality
  - Non-iterative method (determinize all ouputs at once)
    - QBF solving

Iterative Relation Determinization

- Single-output relation
  - For a single-output total relation $R(X, y)$, we derive a function f for variable y using interpolation





$\neg R(X,0) \wedge \neg R(X,1)$ UNSAT

$\varphi_B : \neg R(X,1)$
Minimal care offset of f

$\varphi_A : \neg R(X,0)$
Minimal care onset of f

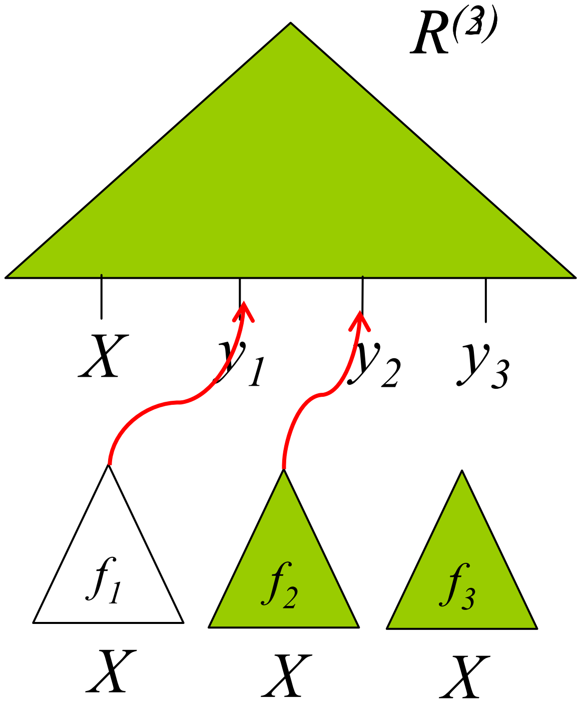
Iterative Relation Determinization

- Multi-output relation
  - Two-phase computation:
    1. Backward reduction
       - Reduce to single-output case

         $R(X, y_1, \ldots, y_n) \rightarrow \exists y_2, \ldots, \exists y_n.\ R(X, y_1, \ldots, y_n)$
    2. Forward substitution
       - Extract functions

Iterative Relation Determinization

- Example



Phase1: (expansion reduction)

$\exists y_3.R(X, y_1, y_2, y_3) \rightarrow R^{(3)}(X, y_1, y_2)$

$\exists y_2.R^{(3)}(X, y_1, y_2) \rightarrow R^{(2)}(X, y_1)$

Phase2:

$R^{(2)}(X, y_1) \rightarrow y_1 = f_1(X)$

$R^{(3)}(X, y_1, y_2) \rightarrow R^{(3)}(X, f_1(X), y_2) \rightarrow y_2 = f_2(X)$

$R(X, y_1, y_2, y_3) \rightarrow R(X, f_1(X), f_2(X), y_2) \rightarrow y_3 = f_3(X)$

Non-Iterative Relation Determinization

- Solve QBF

$$\forall x_1, \ldots, \forall x_m, \exists y_1, \ldots, \exists y_n. R(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

  - The Skolem functions of variables $y_1, \ldots, y_n$ correspond to the functions we want

Summary

- Relation determinization correspond to solving a QBF problem
- Iterative and non-iterative methods can be applied to extract functions from a Boolean relation