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

- HDL spec.
- Logic synthesis
- Logic netlist
- Circuit netlist
- Physical design
- Layout / mask
- Chip
- Fab.
- RTL synthesis
- Fab.
Logic Synthesis

Boolean Function Expression \rightarrow Logic Synthesis \rightarrow Optimized Logic Netlist
Logic Synthesis

Given: Functional description of finite-state machine $F(Q,X,Y,\delta,\lambda)$ where:
- $Q$: Set of internal states
- $X$: Input alphabet
- $Y$: Output alphabet
- $\delta: X \times Q \rightarrow Q$ (next state function)
- $\lambda: X \times Q \rightarrow Y$ (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 $f: B^m \rightarrow Y$, where $B = \{0,1\}$ and $Y = \{0,1,d\}$
  - E.g.
  - The output value of $f(x_1, x_2, x_3)$, say, partitions $B^m$ 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 $f^0 = x_1 \land \neg x_2$)
    - **don’t-care set ($f=d$)**
      - E.g. $\{000, 001\}$ (characteristic function $f^d = \neg x_1 \land \neg x_2$)

- $f$ 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 $f: \mathbb{B}^n \rightarrow \mathbb{B}$ over variables $x_1, \ldots, x_n$ maps each Boolean valuation (truth assignment) in $\mathbb{B}^n$ 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 $f^1 = \{v \in B^n \mid f(v) = 1\}$
  - If $f^1 = B^n$, $f$ is a tautology
- **Offset** of $f$, denoted as $f^0$, is $f^0 = \{v \in B^n \mid f(v) = 0\}$
  - If $f^0 = B^n$, $f$ is unsatisfiable. Otherwise, $f$ is satisfiable.
- $f^1$ and $f^0$ are sets, not functions!
- Boolean functions $f$ and $g$ are **equivalent** if $\forall v \in B^n. f(v) = g(v)$ 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

\[
\begin{align*}
  f(x_1, x_2, x_3) &= x_1 \\
  f(x_1, x_2, x_3) &= \overline{x_1}
\end{align*}
\]
Boolean Function

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

<table>
<thead>
<tr>
<th>$x_1x_2x_3$</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Operations

Given two Boolean functions:

\[ f : \mathbb{B}^n \rightarrow \mathbb{B} \]
\[ g : \mathbb{B}^n \rightarrow \mathbb{B} \]

- \( h = f \land g \) from AND operation is defined as
  \[ h^1 = f^1 \land g^1; \ h^0 = \mathbb{B}^n \setminus h^1 \]

- \( h = f \lor g \) from OR operation is defined as
  \[ h^1 = f^1 \lor g^1; \ h^0 = \mathbb{B}^n \setminus h^1 \]

- \( h = \neg f \) from COMPLEMENT operation is defined as
  \[ h^1 = f^0; \ h^0 = f^1 \]
Cofactor and Quantification

Given a Boolean function:
\[ f : \mathbb{B}^n \rightarrow \mathbb{B}, \text{ with the input variable } (x_1, x_2, \ldots, x_i, \ldots, x_n) \]

- **Positive cofactor on variable** \( x_i \)
  \[ h = f_{x_i} \text{ is defined as } h = f(x_1, x_2, \ldots, 1, \ldots, x_n) \]

- **Negative cofactor on variable** \( x_i \)
  \[ h = f_{\neg x_i} \text{ is defined as } h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \]

- **Existential quantification over variable** \( x_i \)
  \[ h = \exists x_i. f \text{ is defined as } h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \lor f(x_1, x_2, \ldots, 1, \ldots, x_n) \]

- **Universal quantification over variable** \( x_i \)
  \[ h = \forall x_i. f \text{ is defined as } h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \land f(x_1, x_2, \ldots, 1, \ldots, x_n) \]

- **Boolean difference over variable** \( x_i \)
  \[ h = \partial f/\partial x_i \text{ is defined as } h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \oplus f(x_1, x_2, \ldots, 1, \ldots, x_n) \]
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 \( f : \mathbb{B}^n \rightarrow \mathbb{B} \) is a tabulation of its value at each of the \( 2^n \) vertices of \( \mathbb{B}^n \).

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

\[
\text{formula ::= } \begin{align*}
&\text{'}(\text{ formula ')} \\
&\text{ Boolean constant } \quad \text{(true or false)} \\
&\text{ <Boolean variable>} \\
&\text{ formula "+" formula } \quad \text{(OR operator)} \\
&\text{ formula "\cdot" formula } \quad \text{(AND operator)} \\
&\text{ }\neg\text{ formula } \quad \text{(complement)}
\end{align*}
\]

**Example**

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

typically "\cdot" is omitted and '\(\text{'}\), '\)\ are omitted when the operator priority is clear, e.g., \(f = x_1 \cdot x_2 + x_3 + x_4 \cdot \neg x_1\)
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
\[ \phi = ab + a'c + bc \]
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)

\[ f = ab + a'c + a'bd \]
Boolean Function Representation
Binary Decision Diagram

- Any Boolean function $f$ can be written in term of **Shannon expansion**
  \[ f = v f_v + \overline{v} f_{\overline{v}} \]
  - Positive cofactor: $f_{x_i} = f(x_1, \ldots, x_i=1, \ldots, x_n)$
  - Negative cofactor: $f_{\overline{x_i}} = f(x_1, \ldots, x_i=0, \ldots, x_n)$

- 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 $f_{\overline{v}}$
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
      \[ x_{i_1} < x_{i_2} < x_{i_3} < ... < x_{i_n} \]
  - **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

![Diagram showing ordered and unordered BDD examples](image)
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

\[ f = ab + a'c + bc'd \]
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:
  \[ SUPPORT(g) = CONE(g) \cap I \]
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"
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, Sorenson, 2003 (MiniSat)]
Pre-Modern SAT Procedure

Algorithm $\text{DPLL}(\Phi)$
{
    while there is a unit clause $\{l\}$ in $\Phi$
        $\Phi = \text{BCP}(\Phi, l)$;
    while there is a pure literal $l$ in $\Phi$
        $\Phi = \text{assign}(\Phi, l)$;
    if all clauses of $\Phi$ satisfied return true;
    if $\Phi$ has a conflicting clause return false;
    $l := \text{chooseLiteral}(\Phi)$;
    return $\text{DPLL}(\text{assign}(\Phi, \neg l)) \lor \text{DPLL}(\text{assign}(\Phi, l))$;
}
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);
    }
}
There can be many learnt clauses from a conflict.
Clause learning admits non-chronological backtrack.

E.g.,
\{\neg x_{10587}, \neg x_{10588}, \neg x_{10592}\}

\cdots
\{\neg x_{10374}, \neg x_{10582}, \neg x_{10578}, \neg x_{10373}, \neg x_{10629}\}

\cdots
\{x_{10646}, x_{9444}, \neg x_{10373}, \neg x_{10635}, \neg x_{10637}\}
Clause Learning as Resolution

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

$$
\begin{array}{c}
C_1 \lor x \\ \hline \\
C_2 \lor \neg x \\
\end{array}
\Rightarrow 
\frac{C_1 \lor x}{C_1 \lor C_2}
$$

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
    \{-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

\[(a \lor b \lor c)(\neg a \lor c)(\neg b \lor \neg d)(\neg c)(c \lor \neg d)\]

\[\neg d\]

\[()\]
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 $A \land B$ is UNSAT for formulae $A$ and $B$, there exists an interpolant $I$ of $A$ such that

1. $A \Rightarrow I$
2. $I \land B$ 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 $\varphi$.
- For $\varphi = \varphi_A \land \varphi_B$ specified, the corresponding interpolant can be obtained in time linear in the resolution proof.

\[
\varphi_A = (a \lor b \lor c)(\neg a \lor c)(\neg b \lor d)(\neg c)(c \lor d)
\]

\[
\varphi_B = (b \lor c)(c)(1)(1)(1)
\]

\[
\neg d
\]

\[
\land
\]

\[
\lor
\]

\[
= (b \lor c)
\]

[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 $\varphi$?
  - What if deleting a clause from $\varphi$?
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

- If \( f(x) \) functionally depends on \( g_1(x), g_2(x), \ldots, g_m(x) \) if \( f(x) = h(g_1(x), g_2(x), \ldots, g_m(x)) \), denoted \( h(G(x)) \)

- Under what condition can function \( f \) be expressed as some function \( h \) over a set \( G=\{g_1,\ldots,g_m\} \) of functions?

- \( h \) exists \( \iff \exists a, b \) 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
  - ...

![Boolean Network Diagram](image)

- **target function**
- **base functions**
BDD-Based Computation

BDD-based computation of $h$

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

$h_{off} = \{ y \in B^m : y = G(x) \text{ and } f(x) = 0, \ x \in B^n \}$
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 \( \iff \exists a, b \text{ such that } f(a) \neq f(b) \text{ and } G(a) = G(b), \)
i.e., \((f(x) \neq f(x^*)) \land (G(x) \equiv G(x^*))\) is UNSAT

- How to derive \( h \)? How to select \( G \)?
SAT-Based Computation

\[ (f(x) \neq f(x^*)) \land (G(x) \equiv G(x^*)) \] 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 $\text{Img}(f_{on})$ and is disjoint from $\text{Img}(f_{off})$
- $I$ only refers to $y_1, \ldots, y_m$
- Therefore, $I$ corresponds to a feasible implementation of $h$
Incremental SAT Solving

- Controlled equality constraints
  \[(y_i \equiv y_i^*) \rightarrow (\neg y_i \lor y_i^* \lor \alpha_i)(y_i \lor \neg y_i^* \lor \alpha_i)\]
  with auxiliary variables \(\alpha_i\)

- \(\alpha_i = \text{true} \Rightarrow i^{th} \text{ equality constraint is disabled}\)

- 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 $|G|$ 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 $|G|$ different target functions $f$
  - Slow enumeration of different base functions $G$

- Pros
  - All possible implementations of $h$
## Practical Evaluation

### SAT vs. BDD

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#Nodes</th>
<th>#FF</th>
<th>#Dep-S</th>
<th>#Dep-B</th>
<th>#FF</th>
<th>#Dep-S</th>
<th>#Dep-B</th>
<th>Time</th>
<th>Mem</th>
<th>SAT (original)</th>
<th>BDD (original)</th>
<th>SAT (retimed)</th>
<th>BDD (retimed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s5378</td>
<td>2794</td>
<td>179</td>
<td>52</td>
<td>25</td>
<td>398</td>
<td>283</td>
<td>173</td>
<td>1.2</td>
<td>18</td>
<td>1.6</td>
<td>20</td>
<td>0.6</td>
<td>18</td>
</tr>
<tr>
<td>s9234.1</td>
<td>5597</td>
<td>211</td>
<td>46</td>
<td>x</td>
<td>459</td>
<td>301</td>
<td>201</td>
<td>4.1</td>
<td>19</td>
<td>x</td>
<td>x</td>
<td>1.7</td>
<td>19</td>
</tr>
<tr>
<td>s13207.1</td>
<td>8022</td>
<td>638</td>
<td>190</td>
<td>136</td>
<td>1930</td>
<td>802</td>
<td>x</td>
<td>15.6</td>
<td>22</td>
<td>31.4</td>
<td>78</td>
<td>15.3</td>
<td>22</td>
</tr>
<tr>
<td>s15850.1</td>
<td>9785</td>
<td>534</td>
<td>18</td>
<td>9</td>
<td>907</td>
<td>402</td>
<td>x</td>
<td>23.3</td>
<td>22</td>
<td>82.6</td>
<td>94</td>
<td>7.9</td>
<td>22</td>
</tr>
<tr>
<td>s35932</td>
<td>16065</td>
<td>1728</td>
<td>0</td>
<td>--</td>
<td>2026</td>
<td>1170</td>
<td>--</td>
<td>176.7</td>
<td>27</td>
<td>1117</td>
<td>164</td>
<td>78.1</td>
<td>27</td>
</tr>
<tr>
<td>s38417</td>
<td>2397</td>
<td>1636</td>
<td>95</td>
<td>--</td>
<td>5016</td>
<td>243</td>
<td>--</td>
<td>270.3</td>
<td>30</td>
<td>--</td>
<td>--</td>
<td>123.1</td>
<td>32</td>
</tr>
<tr>
<td>s38584</td>
<td>19407</td>
<td>1452</td>
<td>24</td>
<td>--</td>
<td>4350</td>
<td>2569</td>
<td>--</td>
<td>166.5</td>
<td>21</td>
<td>--</td>
<td>--</td>
<td>99.4</td>
<td>30</td>
</tr>
<tr>
<td>b12</td>
<td>946</td>
<td>121</td>
<td>4</td>
<td>2</td>
<td>170</td>
<td>66</td>
<td>33</td>
<td>0.15</td>
<td>17</td>
<td>12.8</td>
<td>38</td>
<td>0.13</td>
<td>17</td>
</tr>
<tr>
<td>b14</td>
<td>9847</td>
<td>245</td>
<td>2</td>
<td>--</td>
<td>245</td>
<td>2</td>
<td>--</td>
<td>3.3</td>
<td>22</td>
<td>--</td>
<td>--</td>
<td>5.2</td>
<td>22</td>
</tr>
<tr>
<td>b15</td>
<td>8367</td>
<td>449</td>
<td>0</td>
<td>--</td>
<td>1134</td>
<td>793</td>
<td>--</td>
<td>5.8</td>
<td>22</td>
<td>--</td>
<td>--</td>
<td>5.8</td>
<td>22</td>
</tr>
<tr>
<td>b17</td>
<td>30777</td>
<td>1415</td>
<td>0</td>
<td>--</td>
<td>3967</td>
<td>2350</td>
<td>--</td>
<td>119.1</td>
<td>28</td>
<td>--</td>
<td>--</td>
<td>161.7</td>
<td>42</td>
</tr>
<tr>
<td>b18</td>
<td>111241</td>
<td>3320</td>
<td>5</td>
<td>--</td>
<td>9254</td>
<td>5723</td>
<td>--</td>
<td>1414</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>2842.6</td>
<td>100</td>
</tr>
<tr>
<td>b19</td>
<td>224624</td>
<td>6642</td>
<td>0</td>
<td>--</td>
<td>7164</td>
<td>337</td>
<td>--</td>
<td>8184.8</td>
<td>217</td>
<td>--</td>
<td>--</td>
<td>11040.6</td>
<td>234</td>
</tr>
<tr>
<td>b20</td>
<td>19682</td>
<td>490</td>
<td>4</td>
<td>--</td>
<td>1604</td>
<td>1167</td>
<td>--</td>
<td>25.7</td>
<td>28</td>
<td>--</td>
<td>--</td>
<td>36</td>
<td>30</td>
</tr>
<tr>
<td>b21</td>
<td>20027</td>
<td>490</td>
<td>4</td>
<td>--</td>
<td>1950</td>
<td>1434</td>
<td>--</td>
<td>24.6</td>
<td>29</td>
<td>--</td>
<td>--</td>
<td>36.3</td>
<td>31</td>
</tr>
<tr>
<td>b22</td>
<td>29162</td>
<td>735</td>
<td>6</td>
<td>--</td>
<td>3013</td>
<td>2217</td>
<td>--</td>
<td>73.4</td>
<td>36</td>
<td>--</td>
<td>--</td>
<td>90.6</td>
<td>37</td>
</tr>
</tbody>
</table>
Practical Evaluation

![Graph showing circuit size vs. runtime]

- R\(^2\) = 0.909
- R\(^2\) = 0.9664
Practical Evaluation

Incremental SAT

Time (log)

Iteration

FLOLAC 2011
Practical Evaluation

#total input vs. #redundant inputs

Number of input variables

Number of spurious variables
Practical Evaluation

Interpolant size vs. support size

Number of variables (log)

Interpolant size (log)

R² = 0.861

R² = 0.8506

Original

Retimed
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 $X_C = \emptyset$, the decomposition is called disjoint; otherwise, non-disjoint
We consider OR, AND, XOR bi-decompositions.

These three cases are sufficient to generate any other type of bi-decomposition.

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a+b</th>
<th>ab</th>
<th>a⊕b</th>
<th>a(¬b)</th>
<th>a(¬b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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:**
  \[ X_C = \emptyset \]

- **Example**
  \[ f(a, b, c, d) = (\neg a)b + cd \]

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

  \[ f(X) = (\neg a)b + cd = f_A(a, b) + f_B(c, d) \]
OR Bi-Decomposition

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

- Example:
  - $f(1101) = 0 = f_A(11) \lor f_B(01)$
  - $f(0010) = 0 = f_A(00) \lor f_B(10)$
  - $f(1110) = 1 = f_A(11) \lor f_B(10)$

<table>
<thead>
<tr>
<th>$X_B \setminus X_A$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f_A(X_A)$</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_B(X_B)$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$X_B \setminus X_A$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f_A(X_A)$</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_B(X_B)$</td>
<td>?</td>
</tr>
</tbody>
</table>
SAT-Based OR Decomposition

\[\exists f_A, f_B \text{ such that } f(X) = f_A(X_A) \lor f_B(X_B)\]

\[\iff \text{For every 1-entry, no 0-entries can appear simultaneously in the corresponding row and column}\]

\[\iff f(X_A, X_B) \land \neg f(X_A', X_B) \land \neg f(X_A, X_B') \text{ is unsatisfiable}\]
SAT-Based OR Decomposition

\[ \exists f_A, f_B \text{ such that } f(X) = f_A(X_A, X_C) \lor f_B(X_B, X_C) \]

⇔ Under every valuation of \( X_C \), for every 1-entry, no 0-entries can appear simultaneously in the corresponding row and column

⇔ \( f(X_A, X_B, X_C) \land \neg f(X_A', X_B, X_C) \land \neg f(X_A, X_B', X_C) \) is unsatisfiable
SAT-Based OR Decomposition

∃f_A, f_B such that \( f(X) = f_A(X_A, X_C) \lor f_B(X_B, X_C) \)
\[ \iff f(X_A, X_B, X_C) \land \neg f(X_A', X_B, X_C) \land \neg f(X_A, X_B', X_C) \text{ is UNSAT} \]

- How to compute \( f_A \) and \( f_B \)? How to determine the variable partition?
SAT-Based OR Decomposition
\( f_A \) Computation

\[ f(X_A, X_B, X_C) \land \neg f(X_A', X_B, X_C) \land \neg f(X_A, X_B', X_C) \text{ is UNSAT} \]
SAT-Based OR Decomposition $f_B$ Computation

$f(X_A, X_B, X_C) \land \neg f_A(X_A, X_C) \land \neg f(X_A', X_B, X_C)$ is UNSAT
SAT-Based OR Decomposition

Variable Partition

\[ \varphi_A = f(X) \land \neg f(X') \land \bigwedge ((x_i \equiv x'_i) \lor \alpha_{x_i}) \]

\[ \varphi_B = \neg f(X'') \land \bigwedge ((x_i \equiv x''_i) \lor \beta_{x_i}) \]

<table>
<thead>
<tr>
<th>(\alpha_x, \beta_x)</th>
<th>( \times ) is belongs to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>( X_c )</td>
</tr>
<tr>
<td>(0,1)</td>
<td>( X_B )</td>
</tr>
<tr>
<td>(1,0)</td>
<td>( X_A )</td>
</tr>
<tr>
<td>(1,1)</td>
<td>either ( X_A ) or ( X_B )</td>
</tr>
</tbody>
</table>
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 \(\alpha x_1 = 0, \beta x_1 = 0, \alpha x_2 = 0, \text{and} \beta x_3 = 0\) causes unsatisfiability. So \(x_1 \in X_C, x_2 \in X_B, \text{and} x_3 \in X_A\)
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 \( f = f_A \land f_B \)
  \[ \iff \exists f_A, f_B \text{ such that } \neg f = \neg f_A \lor \neg f_B \]

- Example

  \[ f(a,b,c,d) = (a + \neg b + c)(b + \neg c + d) \]
  \[ \neg f(a,b,c,d) = (\neg a)b(\neg c) \lor (\neg b)c(\neg d) \]
  \[ = \neg f_A(a,b,c) \lor \neg f_B(b,c,d) \]
  \[ f_A(a,b,c) = (a + \neg b + c), \quad f_B(b,c,d) = (b + \neg c + d) \]
  \[ f(a,b,c,d) = f_A(a,b,c) \land f_B(b,c,d) \]
SAT-Based XOR Decomposition

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

<table>
<thead>
<tr>
<th>(X_B) (X_A)</th>
<th>(X'_B) (X_A')</th>
<th>(X'_B) (X_A')</th>
<th>(f_A(X_A))</th>
<th>(f_B(X_B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_B/X_A)</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>(f_B(X_B))</td>
<td>(f_B(X_B))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(f_B(X_B))</td>
<td>(f_B(X_B))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(f_A(X_A)\) (5) (6)

\(X_A\) \(X_C\) \(X_B\)
SAT-Based XOR Decomposition

- $[(1) \equiv (2)] \land [(3) \not\equiv (4)] \text{ UNSAT}$
- $\exists f_A, f_B$ such that $f(X) = f_A(X_A, X_C) \oplus f_B(X_B, X_C) \iff (f(X_A, X_B, X_C) \equiv f(X_A, X_B', X_C)) \land (f(X_A', X_B, X_C) \not\equiv f(X_A', X_B', X_C))$ \text{ UNSAT}

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

<table>
<thead>
<tr>
<th>$X_B \setminus X_A$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$X_B' \setminus X_A$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**SAT-Based XOR Decomposition**

$f_A, f_B$ Computation

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

<table>
<thead>
<tr>
<th>$X_c$</th>
<th>$X_B \setminus X_A$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>$f_B(X_B, X_C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$f_A(X_A, X_C)$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
SAT-Based XOR Decomposition

Variable Partition

- Similar to OR decomposition

\[(f(X) \equiv f(X')) \land (f(X'') \neq f(X''')) \land \\
((x_i \equiv x_i'') \land (x_i' \equiv x_i'''')) \lor \alpha_{x_i} \land \\
((x_i \equiv x_i') \land (x_i'' \equiv x_i''')) \lor \beta_{x_i})\]

<table>
<thead>
<tr>
<th>((\alpha_{x_i}, \beta_{x_i}))</th>
<th>(X_i) belongs to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(X_C)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(X_B)</td>
</tr>
<tr>
<td>(1,0)</td>
<td>(X_A)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>either (X_A) or (X_B)</td>
</tr>
</tbody>
</table>
### Practical Evaluation

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

\[ Q_1 x_1, \ldots, Q_n x_n. \varphi \]

for \( Q_i \in \{ \forall, \exists \} \) and \( \varphi \) a quantifier-free formula

- If \( \varphi \) 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 \( \neg a \neg bc \neg d \) (can be further simplified to \( \neg a \))
QBF Solving

Example
\[ \exists a \forall x \forall b \forall y \exists c \ (a + b + y + c)(a + x + b + y + c)(x + b)(y + c)(c + a + x + b)(x + b)(a + b + y) \]

\[ <a, L > \]
\[ (b + y + c)(x + b + y + c)(x + b)(y + c)(x + b)(b + y) \]

\[ < x, L > \]
\[ (b + y + c)(b + y + c)(b)(y + c)(b + y) \]

\[ < x, R > \]
\[ (b + y + c)(y + c)(b) \]

\[ < b, U > \]
\[ (y + c)(y + c)(y + c) \]

\[ < b, U > \]
\[ (x + b)(c + x + b)(x + b) \]

\[ < c, U > \]
\[ (x + b)(x + b)(x + b) \]

\[ < y, P > \]
\[ (x + b)(y + c)(c + x + b)(x + b) \]

\[ < y, P > \]
\[ (x + b)(y + c)(x + b)(x + b) \]

\[ < c, P > \]
\[ (x + b)(x + b)(x + b) \]

\[ < x, L > \]
\[ (c)(c) \]

\[ < y, L > \]
\[ (c)(c) \]

\[ {\text{true}} \]
\[ {\text{false}} \]

\[ {\text{true}} \]
\[ {\text{false}} \]

\[ {\text{true}} \]
\[ {\text{false}} \]

\[ \exists \]
\[ \forall \]

\[ \{axbc\} \]

\[ \{axbyc\} \]

\[ \{axbc\} \]

\[ \{false\} \]

FLOLAC 2011
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**

\[
\begin{align*}
C_1 \lor x & \quad C_2 \lor \neg x \\
\hline
\forall-RED(C_1 \lor C_2)
\end{align*}
\]

where operator \(\forall-RED\) removes from \(C_1 \lor C_2\) the universally (\(\forall\)) quantified variables whose quantification levels are greater than any of the existentially (\(\exists\)) quantified variables in \(C_1 \lor C_2\)

- E.g.,
  - prefix: \(\forall a \exists b \forall c \forall d \exists e\)
  - \(\forall-RED(a+b+c+d) = (a+b)\)

- 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)

\[ \exists a \forall x \exists b \forall y \exists c \quad (a+b+y+c)(a+x+b+y+c)(x+b)(y+c)(c+a+x+b)(x+b)(a+b+y) \]

\[ \begin{align*}
\langle \bar{a}, L \rangle & \quad \langle a \rangle \\
\langle x, L \rangle & \quad \langle a \rangle \\
\langle x+b \rangle & \quad \langle b, U \rangle \\
\langle \bar{y}, L \rangle & \quad \langle a+b+x \rangle \\
\langle \bar{c}, L \rangle & \quad \langle a+b+y+c \rangle \\
\langle c, R \rangle & \quad \langle a+b+x+y+c \rangle \\
\langle \bar{b}, L \rangle & \quad \langle c+a+x+b \rangle \\
\langle b, R \rangle & \quad \langle x+b \rangle \\
\end{align*} \]
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

<table>
<thead>
<tr>
<th>Solver</th>
<th>Algorithm</th>
<th>Certificate True QBF</th>
<th>Certificate False QBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuBE-cert</td>
<td>search</td>
<td>Cube resolution</td>
<td>Clause resolution</td>
</tr>
<tr>
<td>yQuaffle</td>
<td>search</td>
<td>Cube resolution</td>
<td>Clause resolution</td>
</tr>
<tr>
<td>Ebddres</td>
<td>Skolemization</td>
<td>Skolem function</td>
<td>Clause resolution</td>
</tr>
<tr>
<td>sKizzo</td>
<td>Skolemization</td>
<td>Skolem function</td>
<td>-</td>
</tr>
<tr>
<td>squolem</td>
<td>Skolemization</td>
<td>Skolem function</td>
<td>Clause resolution</td>
</tr>
</tbody>
</table>
QBF Certification

- Incomplete picture of QBF certification

<table>
<thead>
<tr>
<th></th>
<th>Syntactic Certificate</th>
<th>Semantic Certificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>True QBF</td>
<td>Cube-resolution proof</td>
<td>Skolem-function model</td>
</tr>
<tr>
<td>False QBF</td>
<td>Clause-resolution proof</td>
<td>?</td>
</tr>
</tbody>
</table>

- 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

True QBF
- Cube resolution proof
  - ResQu
- Skolem function (model)

False QBF
- Clause resolution proof
  - ResQu
- Herbrand function (countermodel)

Formal negation
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.

\[ \varphi := \text{clause} \mid \text{cube} \mid \text{clause} \land \varphi \mid \text{cube} \lor \varphi \]

E.g.,

\[(a'+b) \land ac \lor (b'+c') \land bc = ((a'+b) \land (ac \lor ((b'+c') \land bc)))\]
ResQu

**Countermodel** _construct_

**input:** a false QBF $\Phi$ and its clause-resolution DAG $G_H(V_H, E_H)$

**output:** a countermodel in RFAO formulas

**begin**

01 **foreach** universal variable $x$ of $\Phi$
02 $\text{RFAO\_node\_array}[x] := \emptyset$;
03 **foreach** vertex $v$ of $G_H$ in topological order
04 if $v$.clause resulted from $\forall$-reduction on $u$.clause, i.e., $(u, v) \in E_H$
05 $v$.cube := $\neg(v$.clause$)$;
06 **foreach** universal variable $x$ reduced from $u$.clause to get $v$.clause
07 if $x$ appears as positive literal in $u$.clause
08 push $v$.clause to $\text{RFAO\_node\_array}[x]$;
09 else if $x$ appears as negative literal in $u$.clause
10 push $v$.cube to $\text{RFAO\_node\_array}[x]$;
11 if $v$.clause is the empty clause
12 **foreach** universal variable $x$ of $\Phi$
13 simplify $\text{RFAO\_node\_array}[x]$;
14 return $\text{RFAO\_node\_array}$’s;

**end**
Example

\[\exists a \forall x \exists b \forall y \exists c\]

\[(a + b + y + c)_1 (a + x + b + y + c)_2 (x + b)_3 (y + c)_4 (\neg a + x + b + c)_5 (x + b)_6 (a + b + y)_7\]

1. \[(a + b + y)_8\]
2. \[(a + x + b + y)_8^+\]
3. \[(a + x)_9\]
4. \[(a + x + b)_10\]
5. \[(a + x + b)_10^+\]
6. \[(a + x)_11\]
7. \[(a + b)_7^+\]

0. \[x: []; y: []\]
1. \[x: []; y: [\text{cube}(\overline{ab})]\]
2. \[x: []; y: [\text{cube}(\overline{ab}), \text{clause}(a + x + b)]\]
3. \[x: [\text{clause}(a)]; y: [\text{cube}(\overline{ab}), \text{clause}(a + x + b)]\]
4. \[x: [\text{clause}(a)]; y: [\text{cube}(\overline{ab}), \text{clause}(a + x + b), \text{cube}(ax\overline{b})]\]
5. \[x: [\text{clause}(a), \text{cube}(a)]; y: [\text{cube}(\overline{ab}), \text{clause}(a + x + b), \text{cube}(ax\overline{b})]\]

(empty)
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

\[
y_1 y_2 y_3 = f(x_1 x_2 x_3) = g(y_1 y_2 y_3, x_1 x_2 x_3)
\]
Introduction

Motivations

- Theoretically
  - Complexity in between \text{coNP} (for all ... ) and \text{\Sigma}_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

<table>
<thead>
<tr>
<th>Method</th>
<th>Complete</th>
<th>Function type</th>
<th>Equivalence type</th>
<th>Solution type</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral methods</td>
<td>yes</td>
<td>CS</td>
<td>mostly P</td>
<td>one</td>
<td>--</td>
</tr>
<tr>
<td>Signature based methods</td>
<td>no</td>
<td>mostly CS</td>
<td>P/NP</td>
<td>N/A</td>
<td>~ ++</td>
</tr>
<tr>
<td>Canonical-form based methods</td>
<td>yes</td>
<td>CS</td>
<td>mostly P</td>
<td>one</td>
<td>+</td>
</tr>
<tr>
<td>SAT based methods</td>
<td>yes</td>
<td>CS</td>
<td>mostly P</td>
<td>one/all</td>
<td>+</td>
</tr>
<tr>
<td>BooM (QBF/SAT-like)</td>
<td>yes</td>
<td>CS / IS</td>
<td>NPN</td>
<td>one/all</td>
<td>++</td>
</tr>
</tbody>
</table>

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:

\[ O(2^n!) \quad O(2^{2n}) \]
Formulation

- Reduce NPN-equiv to 2 NP-equiv checks
  - Matching $f$ and $g$; matching $f$ and $\neg g$

- 2nd order formula of NP-equivalence
  $$\exists \nu \circ \pi, \forall x \ ((f_c(x) \land g_c(\nu \circ \pi(x))) \Rightarrow (f(x) \equiv g(\nu \circ \pi(x))))$$
  - $f_c$ 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 $\lor \circ \pi$

$$
\begin{bmatrix}
\begin{array}{cc}
  x_1 & \lnot x_1 \\
  a_{11} & b_{11}
\end{array}
\end{bmatrix}
\begin{bmatrix}
\begin{array}{cccc}
  x_2 & \lnot x_2 & \cdots & x_n & \lnot x_n \\
  a_{12} & b_{12} & \cdots & a_{1n} & b_{1n}
\end{array}
\end{bmatrix}
\begin{bmatrix}
\begin{array}{c}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n
\end{array}
\end{bmatrix}
= 1
$$

$\sum = 1$

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

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

- Quantified Boolean formula (QBF) for NP-equivalence
  \[ \exists a, \exists b, \forall x, \forall y \ (\varphi_C \land \varphi_A \land ((f_c \land g_c) \implies (f \equiv g))) \]
  - \( \varphi_C \): cardinality constraint
  - \( \varphi_A \): \( \land_{i,j} (a_{ij} \implies (y_i \equiv x_j)) \) (\( b_{ij} \implies (y_i \equiv \neg x_j) \))

- Look for an assignment to a- and b-variables that satisfies \( \varphi_C \) and makes the miter constraint
  \[ \Psi = \varphi_A \land (f \neq g) \land f_c \land g_c \]
  unsatisfiable

- Refine \( \varphi_C \) iteratively in a sequence \( \Phi^{(0)}, \Phi^{(1)}, \ldots, \Phi^{(k)} \), for \( \Phi^{(i+1)} \implies \Phi^{(i)} \) through conflict-based learning
BooM Flow

- **f (and \( f_c \))**
- **g (and \( g_c \))**

**Preprocess** (sig., abs.)

- **\( \Phi^{(i)} \) characterizes all matches**

**Solve** \( \Phi^{(i)} \land \Psi \)

- **SAT?**
  - no: **No match**
  - yes: **Add learned clause to** \( \Phi^{(i)} \)

**Solve miter** \( \Psi \)

- **SAT?**
  - no: **no**
  - yes: **yes**

**How to compute all matches?**
NP-Equivalence
Conflict-based Learning

Observation

How to avoid these 6 mappings at once?

From SAT 1
NP-Equivalence
Conflict-based Learning

- Learnt clause generation

\(( a_{11} \lor b_{12} \lor a_{13} \lor b_{21} \lor a_{22} \lor b_{23} \lor b_{31} \lor a_{32} \lor b_{33} )\)
NP-Equivalence
Conflict-based Learning

- Proposition:
  If \( f(u) \neq g(v) \) with \( v = v \circ \pi(u) \) for some \( v \circ \pi \) satisfying \( \Phi^{(i)} \), then the learned clause \( \bigvee_{ij} l_{ij} \) for literals
  \( l_{ij} = (v_i \neq u_j) \land a_{ij} : b_{ij} \)
  excludes from \( \Phi^{(i)} \) the mappings \( \{ v' \circ \pi' \mid v' \circ \pi'(u) = v \circ \pi(u) \} \)

- Proposition:
  The learned clause prunes \( n! \) infeasible mappings

- Proposition:
  The refinement process \( \Phi^{(0)}, \Phi^{(1)}, \ldots, \Phi^{(k)} \) 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

\[
\begin{pmatrix}
\sum_y \begin{pmatrix}
a_{11} & b_{11} & \cdots & a_{1k} & b_{1k} \\
a_{21} & b_{21} & \cdots & a_{2k} & b_{2k} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
a_{n1} & b_{n1} & \cdots & a_{nk} & b_{nk}
\end{pmatrix} & a_{1(k+1)} & b_{1(k+1)} \\
& a_{2(k+1)} & b_{2(k+1)} \\
& a_{n(k+1)} & b_{n(k+1)}
\end{pmatrix} = 1
\]

- 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 \( \pi \) satisfying \( \Phi^{\langle i \rangle} \), then the learned clause \( \lor_{ij} l_{ij} \) for literals
\[
l_{ij} = (v_i=0 \text{ and } u_j=1) \ ? \ a_{ij} : \emptyset
\]
excludes from \( \Phi^{\langle i \rangle} \) the mappings \( \{ \pi' \mid \pi'(u) = \pi(u) \} \)
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)

FLOLAC 2011
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 $O(2^{n!})$ to $O(2^{2n})$
      - Empirically exponent $\sim 7$ times less for P, $\sim 3$ 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

\[
\begin{array}{cccc}
 x_1 & x_2 & y_1 & y_2 \\
00 & 00 & 00 & \\
01 & 01 & 01 & \\
10 & 10 & 10 & \\
11 & 11 & 11 & \\
\end{array}
\]

\[
\begin{array}{cccc}
 x_1 & x_2 & y_1 & y_2 \\
00 & 00 & 00 & \\
01 & 01 & 01 & \\
10 & 10 & 10 & \\
11 & 11 & 11 & \\
\end{array}
\]

\[
f_1 = x_1 \cdot x_2 \\
f_2 = \neg x_1 \cdot \neg x_2
\]
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

\[
\begin{array}{ccc}
00 & 01 & 10 \\
\downarrow & \downarrow & \downarrow  \\
\circ & \circ & \circ \\
10 & 11 & 00 \\
\end{array}
\]

\[
\begin{array}{ccc}
00 & 01 & 10 \\
\downarrow & \downarrow & \downarrow  \\
\circ & \circ & \circ \\
10 & 11 & 00 \\
\end{array}
\]
A partial relation can be **totalized**

Assume that the input element not mapped to any output element is a don’t care

\[ T(X, y) = R(X, y) \lor \forall y. \neg R(X, y) \]
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

\[ f_1 = x_1 x_2 \]
\[ f_2 = \overline{x}_1 \overline{x}_2 \]
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 outputs 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.

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

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

$\rightarrow R(X, 0) \land \neg R(X, 1)$ UNSAT
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

Phase 1: (expansion reduction)

\[ \exists y_3. R(X, y_1, y_2, y_3) \rightarrow R^{(3)}(X, y_1, y_2) \]
\[ \exists y_2. R^{(3)}(X, y_1, y_2) \rightarrow R^{(2)}(X, y_1) \]

Phase 2:

\[ R^{(2)}(X, y_1) \rightarrow y_1 = f_1(X) \]
\[ R^{(3)}(X, y_1, y_2) \rightarrow R^{(3)}(X, f_1(X), y_2) \rightarrow y_2 = f_2(X) \]
\[ R(X, y_1, y_2, y_3) \rightarrow R(X, f_1(X), f_2(X), y_2) \rightarrow y_3 = f_3(X) \]
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