1. Logic Verification

- Basic principles of OBDD’s
- Variable ordering
- Network of gates \( \Rightarrow \) OBDD’s
- FDD’s and OKFDD’s
- Reasoning about circuits
- Structural methods
- Satisfiability checker

The basic problem: prove that two circuits implement the same boolean function, i.e., that \( f \equiv g \) is a tautology

\[ f \equiv g = f \land g + f \land \overline{g} \]
1. Logic verification

Basic principles of OBDDs

- Ordered Binary Decision Diagrams
- Decision diagrams:
  - one of many possible representations of boolean functions
  - based on Boole’s expansion theorem (1849)
  - seminal paper by Bryant (1986) on the application to logic verification

Principle: "Divide and conquer"

f(0, b, c, d)
f(1, b, c, d)
two subfunctions that do not depend on variable a
Notation:

- \( f(0, b, c, d) \) and \( f(1, b, c, d) \) are the cofactors of \( f \) w.r.t. \( a \)
- we write also \( f_\bar{a} \) and \( f_a \), respectively
- Boole´s expansion theorem:

\[
f = \bar{a}^* f_\bar{a} + a^* f_a
\]
Apply Boole’s theorem to all variables: decision tree
➤ example: XOR in three variables

Variable ordering: order of application of Boole’s theorem
Observation: there are identical subtrees

Sharing of subtrees => decision graph
Shannon: A symbolic analysis of relay and switching circuits (1938)

Some simple examples of decision diagrams:

- For $n$ odd:
  \[ \sum_{k=1}^{n} x_k \]
- For $n$ even:
  \[ (\sum_{k=1}^{n} x_k)' \]

- $a + b$

- $a * b$
**1. Logic verification**

- **AND, OR, XOR in n variables**

  ![Logic Verification Diagram](image)

  - #nodes grows linearly for AND, OR and XOR

- **Many types of decision diagrams**

  - **most favoured and successful OBDD’s (Ordered Binary Decision Diagrams, Bryant ‘86)**

  - **properties of OBDD’s:**
    - same variable ordering on all paths ("ordered")
      - associate an index index(x) with each variable x
      - if var(v) is the variable associated with node v
      - then index(var(v)) is smaller than the index of all successor nodes
    - "reduced":
      - there are no two nodes that represent the same function
      - the two successors of each node are not identical

---

1-7  
**Tutorial on Formal Verification**
Given a variable ordering, OBDD’s are canonical representations of boolean functions.

Two circuits implement the same boolean function $\iff$ the two OBDD’s are identical.
1. Logic verification

○ SN 74181 ALU:

○ SN 74181 OBDD ("shared OBDD" for several outputs):
1. Logic verification

Implementation 1:

Implementation 2:

OBDD’s

Combination of synthesis, extraction, verification (BULL, AT&T, ...):

Specification

VHDL-Description

Synthesis

Verification

OBDD = OBDD

Synthesis

Transistor Netlist

Extraction

Synthesis

Layout

Extraction

Transistor Netlist
OBDD’s become even more compact, if inverted edges are provided. Example:

Variable ordering

* #nodes depends *critically* on the variable ordering

 classical example (Bryant ’86):

\[ f = x_1 x_2 + x_3 x_4 + x_5 x_6 \]
n-bit adder:
- variable ordering R₁: aₙ, bₙ, aₙ₋₁, bₙ₋₁,..., a₀, b₀
- variable ordering R₂: aₙ, aₙ₋₁,..., a₀, bₙ, bₙ₋₁,..., b₀

<table>
<thead>
<tr>
<th>n=</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>0.02</td>
<td>0.03</td>
<td>0.11</td>
<td>0.19</td>
</tr>
<tr>
<td>#nodes</td>
<td>35</td>
<td>75</td>
<td>155</td>
<td>315</td>
</tr>
<tr>
<td>R₂:</td>
<td>time</td>
<td>0.39</td>
<td>16.34</td>
<td></td>
</tr>
<tr>
<td>#nodes</td>
<td>750</td>
<td>196574</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heuristics to determine a "good" variable ordering
- example: distribution of a "weight"

- sum of weights: x=1/2, y=1/4, z=1/4,
  => develop for x first
➤ delete variable and iterate

sum now: \( y = \frac{3}{4}, z = \frac{1}{4}, \)

\( \Rightarrow \) \( y \) is second variable

➤ hence variable ordering \( x, y, z \)

❖ Sifting: dynamic variable ordering (Rudell ICCAD’93)

➤ basic step: exchange adjacent variables (Fujita et al. EDAC’91)
1. Logic verification

➤ Principle: exchange 0-1 and 1-0 path

Sifting: dynamic variable ordering
➤ basic step: exchange adjacent variables
Sifting procedure:
- find variable with max. #nodes (the "thickest" part of an OBDD)
- shift variable over OBDD by pairwise exchange of adjacent variables until #nodes becomes minimum
1. Logic verification

Logic verification 33

Logic verification 34
1. Logic verification

In detail:

- Diagrams showing logic verification processes with nodes labeled V3, V4, V5, and V6, illustrating the flow and connections between these nodes.
1. Logic verification

Logic verification 41

Logic verification 42
1. Logic verification

Logic verification 53

Logic verification 54
Network of gates => OBDD's

Logic verification 57

Logic verification 58
 Traverse network from inputs to outputs + build OBDD's

a
0 1
0 1

b
0 1
0 1

C-program Traverser

C-program &

C-program ≥1

C-program ≥1

 Traverse network from inputs to outputs + build OBDD's

a
0 1
0 1

b
0 1
0 1

C-program Traverser

C-program &

C-program ≥1

C-program ≥1
Traverse network from inputs to outputs + build OBDD's

How does, e.g., the C-program & work?

basis: orthogonality of Boolean expansion, i.e.,

\[ f + g = x^*(f_x + g_x) + \bar{x}^*(f_{\bar{x}} + g_{\bar{x}}), \]
\[ f^* g = x^*(f_x * g_x) + \bar{x}^*(f_{\bar{x}} * g_{\bar{x}}), \]
\[ \tilde{f} = x^*f_x + \bar{x}^*f_{\bar{x}} \]
The AND-Operation between two OBDD's \( \text{bdd1} \) and \( \text{bdd2} \)

- assume nodes of form \((x,v0,v1)\)
- \( \text{var} \) low high

```plaintext
function AND(bdd1, bdd2):

IF \( \text{bdd1} = 0 \) OR \( \text{bdd2} = 0 \) THEN return 0;
ELSEIF \( \text{bdd1} = 1 \) THEN return \( \text{bdd2} \);
ELSEIF \( \text{bdd2} = 1 \) THEN return \( \text{bdd1} \);
ELSE var1 := var(bdd1); var2 := var(bdd2);
   IF var1 = var2 THEN x := var1; v0 := AND(low(bdd1), low(bdd2)), v1 := AND(high(bdd1), high(bdd2));
   ELSEIF index(var1) < index(var2) THEN x := var1; v0 := AND(low(bdd1), \( \text{bdd2} \)), v1 := AND(high(bdd1), \( \text{bdd2} \));
   ELSEIF ...  
   IF v0 = v1 THEN return v0 ELSE return \((x,v0,v1)\); ... 
```

The AND-Operation between two OBDD's \( \text{bdd1} \) and \( \text{bdd2} \)

- assume nodes of form \((x,v0,v1)\)
- \( \text{var} \) low high

![Diagram of OBDD nodes and variables](image)

```plaintext
bdd1 
var1=a 

bdd2 
var2=c  
=>  index(var1) < index(var2)
```
$\text{var1}=a \quad \text{var2}=c \quad \Rightarrow \quad \text{index(var1)} < \text{index(var2)}$
1. Logic verification

```
var1 = b  var2 = c  =>  index(var1) < index(var2)
```

```
x := var1 := b
v0 := and(low(bdd1), bdd2),
v1 := and(high(bdd1), bdd2)
```
1. Logic verification

**Logic Verification Tutorial on Formal Verification**

- **bdd1**
- **bdd2**

<table>
<thead>
<tr>
<th>var1=b</th>
<th>var2=c =&gt; index(var1) &lt; index(var2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=var1 := b</td>
<td></td>
</tr>
<tr>
<td>v0:= and(low(bdd1),bdd2), ✓</td>
<td></td>
</tr>
<tr>
<td>v1:= and(high(bdd1),bdd2) ✓</td>
<td></td>
</tr>
</tbody>
</table>

- **bdd1**
- **bdd2**

<table>
<thead>
<tr>
<th>var1=a</th>
<th>var2=c =&gt; index(var1) &lt; index(var2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=var1 := a</td>
<td></td>
</tr>
<tr>
<td>v0:= and(low(bdd1),bdd2), ✓</td>
<td></td>
</tr>
<tr>
<td>v1:= and(high(bdd1),bdd2) ✓</td>
<td></td>
</tr>
</tbody>
</table>
$\text{var1} = a \quad \text{var2} = c \Rightarrow \text{index(var1)} < \text{index(var2)}$

$x := \text{var1} := a$
$v_0 := \text{and(low(bdd1), bdd2)}$
$v_1 := \text{and(high(bdd1), bdd2)}$

$bdd1 \quad bdd2$

$\text{var1} = a \quad \text{var2} = c \Rightarrow \text{index(var1)} < \text{index(var2)}$

$x := \text{var1} := a$
$v_0 := \text{and(low(bdd1), bdd2)}$
$v_1 := \text{and(high(bdd1), bdd2)}$
#### Problem:

\[ a \oplus b \oplus c \oplus d \oplus e \oplus f \oplus g \]

\[ a \oplus b \oplus c \oplus d \oplus e \oplus f \oplus g \]
"OBDD-packages" maintain two tables:

- the computed table ct has entries of the form

<table>
<thead>
<tr>
<th>Operation</th>
<th>bdd1</th>
<th>bdd2</th>
<th>Result bdd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct stores results calculated before</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- the unique table ut has entries of the form

<table>
<thead>
<tr>
<th>x</th>
<th>v0</th>
<th>v1</th>
</tr>
</thead>
</table>

function **AND**(bdd1, bdd2):

- \( \text{IF } (\text{AND}, \text{bdd1}, \text{bdd2}, x) \in \text{ct} \) \text{ THEN return } x; 
- \( \text{IF } \text{bdd1}=0 \) \text{ OR } \text{bdd2}=0 \text{ THEN return } 0; 
- \( \text{ELSEIF } \text{bdd1}=1 \) \text{ THEN return } bdd2; 
- \( \text{ELSEIF } \text{bdd2}=1 \) \text{ THEN return } bdd1; 
- \( \text{ELSE} \) \text{ var1:=var(bdd1); var2:=var(bdd2); 
- \( \text{IF } \text{var1=var2} \text{ THEN } x:=\text{var1}; \text{ v0:= AND}(\text{low(bdd1)}, \text{low(bdd2)}), \text{ v1:= AND}(\text{high(bdd1)}, \text{high(bdd2)}); 
- \( \text{ELSEIF } \text{index(var1)} < \text{index(var2)} \text{ THEN } x:=\text{var1}; \text{ v0:= AND}(\text{low(bdd1)}, \text{bdd2}), \text{ v1:= AND}(\text{high(bdd1)}, \text{bdd2}); 
- \( \text{ELSEIF } ... 
- \( \text{IF } \text{v0 = v1} \) \text{ THEN return } v0 
- \( \text{ELSEIF } (x,v0,v1) \notin \text{ut} \) \text{ THEN put in ut;} \text{ ELSE return } (x,v0,v1); ...
the result OBDD of a boolean operation of two
OBDD´s of size m and n nodes, respectively, has not
more than n*m nodes
complexity of boolean operations between two
OBDD´s is \( O(n^*m) \)

Many OBDD-packages in the public domain
in many cases based on the \( \text{ite}(p, f, g) \)-operator (if \( p \) then \( f \) else \( g \))
very efficient: the CUDD package from Boulder
see also the TUD DD-package home page by Stefan Höreth
with on-line demo´s of, e.g., sifting
http://www.rs.e-technik.th-darmstadt.de/~sth/

the OBDD technique is a very efficient
decision procedure for the propositional
calculus and incorporated into many theorem
provers, e.g., PVS and ACL2
FDD’s and OKFDD’s

- FDD’s (Functional Decision Diagrams, Kebschull et al. 92)
  - \( f = f_x \oplus x^*(f_{\bar{x}} \oplus f_x) \)
  - for \( x = 0 \) we get \( f_{\bar{x}} \)
  - for \( x = 1 \) we get \( f_{\bar{x}} \oplus f_x \oplus f_x = f_x \)
  - same graph structure, distinct interpretation:

![Diagram of FDD and OKFDD]

- FDD’s are canonical representations
  - fixed variable ordering
  - reduction rule:

- FDD’s are canonical representations
  - fixed variable ordering
  - reduction rule: \( (f_{\bar{x}} \oplus f_x) : \) if the boolean difference is 0, then the function does not depend on \( x \).
Difference between XOR and AND for FDD's:

\[ f \oplus g = f_x \oplus x^*(f_x \oplus f_y) \oplus g_x \oplus x^*(g_x \oplus g_y) = (f_x \oplus g_x) \oplus x^*((f_x \oplus f_y) \oplus (g_x \oplus g_y)) \]

\[ f \ast g = (f_x \oplus x^*(f_x \oplus f_y)) \ast (g_x \oplus x^*(g_x \oplus g_y)) = (f_x g_x) \oplus x^*(f_x (g_x \oplus g_y) \oplus (f_x \oplus f_y) g_x \oplus (f_x \oplus f_y) (g_x \oplus g_y)) \]

For the calculation of the AND, all 4 combinations of high and low successors have to be considered.

OBDD and FDD for 4-bit adder (both with inverted edges)
Three types of decomposition:

- **Shannon:** \( f = \bar{x}f_x + x\bar{f}_x \)
- **positive Davio:** \( f = f_x \oplus x^*(f_x \oplus f_x) \)
- **negative Davio:** \( f = f_x \oplus \bar{x}^*(f_x \oplus f_x) \)

\[ f = a^*[0\oplus c^*(1\oplus 0)] \oplus b^*[1\oplus(0\oplus c^*(1\oplus 0))] + \] \[ \bar{a}^*[0\oplus c^*(1\oplus 0)] = a^*(c\oplus b\bar{c}) + \bar{a}^*c \]

- **OBDD**s:
  - AND, OR, XOR of two OBDD’s of size \( n \) and \( m \) of complexity \( O(n \times m) \)

- **FDD**s/OKFDD**s:
  - XOR of complexity \( O(n \times m) \), but AND and OR exponential
  - #nodes of FDD’s/OKFDD’s may be < #nodes of OBDD’s => synthesis applications
  - OKFDD’s: determining the decomposition-type list (DTL) is an additional problem
Reasoning about circuits

- A circuit with n inputs and m outputs can be modelled as a vector of boolean functions, $F : B^n \rightarrow B^m$
- reasoning about circuits is facilitated if the characteristic function of such a circuit is built
- let $R$ be a subset of $B^n$, $R \subseteq B^n$. Then the characteristic function of this set, $\chi_R : B^n \rightarrow B$, is defined by:

$$\chi_R(x) = \begin{cases} 
1 & \text{if } x \in R \\
0 & \text{if } x \notin R
\end{cases}$$

- The characteristic function $\chi_C$ of a circuit

$$\chi_C = (x \equiv a\cdot b) \land (y \equiv a + b)$$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
<th>y</th>
<th>$\chi_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$$r \equiv s = r^*s + \bar{r}^*s$$
1. Logic verification

The characteristic function $\chi_i$ of all circuit output-values (the "image" of $B^n \rightarrow B^m$)

- e.g., the combination $x=1$ and $y=0$ is not possible

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
<th>y</th>
<th>$\chi_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 ✓</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 ✓</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 ✓</td>
</tr>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 ✓</td>
</tr>
</tbody>
</table>

$x = a \cdot b$, $y = a + b$

$\chi_C = (x \equiv a \cdot b)(y \equiv a + b)$

The existential quantification of several inputs means to build the sum of all input combinations.

Hence, $\chi_i = \exists a, b: (x \equiv a \cdot b)(y \equiv a + b)$

$\chi_i = x \cdot y$

$\chi_C$

$\chi_i = \bar{x} + y$

How to calculate $\chi_i$?

- if $a = 1$ and $b = 0$

$\chi_C = (x \equiv a \cdot b)(y \equiv a + b) = (x \equiv 0)(y \equiv 1) = \bar{x} \cdot y$, i.e., the characteristic function of the output values!

The existence of a, b:

$\chi_i = \exists a, b: (x \equiv a \cdot b)(y \equiv a + b)$

$\chi_i = \bar{x} + y$

$\chi_C$
3 basic operations between cofactors:

- **existential quantification**
  \[ \exists x : f(x) = f_x + f_{\overline{x}} \]

- **universal quantification**
  \[ \forall x : f(x) = f_x \cdot f_{\overline{x}} \]

- **QBF’s: quantified boolean formulas**

  Note:
  \[ \exists x, y : f(x, y) = f(0,0) + f(0,1) + f(1,0) + f(1,1) \]
  i.e., existential quantification of a number of boolean variables means: build the sum of the function-values for all combinations of variables

- **boolean difference**
  \[ \partial f(x)/\partial x = f_x \oplus f_{\overline{x}} \]

How can we characterize all values of x and y for a=1 ("image calculation under a restriction")?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
<th>y</th>
<th>( \chi_C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \mathbb{1} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( \mathbb{0} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \mathbb{1} )</td>
</tr>
</tbody>
</table>
| 1 | 0 | 0 | 1 | \( \checkmark \)
| 1 | 0 | 1 | 0 | \( \checkmark \)
| 1 | 0 | 1 | 1 | \( \checkmark \)
| 1 | 1 | 0 | 0 | \( \mathbb{0} \) |
| 1 | 1 | 0 | 1 | \( \checkmark \)
| 1 | 1 | 1 | 0 | \( \mathbb{0} \) |
| 1 | 1 | 1 | 1 | \( \checkmark \)
How can we characterize all input values of a and b so that x=0 and y=1 ("pre-image calculation")?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
<th>y</th>
<th>χ(_C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(\(\bar{x} \cdot \bar{y}\))[x->a \cdot b, y->a+b]=

(\(a^\ast b\text{')}(a+b)=\bar{a} \cdot b + a \cdot \bar{b}\)

= \exists \ x, y : ((x \equiv a \cdot b)(y \equiv a + b) \ast (\bar{x} \cdot y))

Function substitution: substitute function g vor variable x

\[ f[x->g] = \bar{g} \ast f_x + g \ast f_x \]

Note: \exists \ x : (f*(x \equiv g)) = [f_x^* (0 \equiv g)] + [f_x^* (1 \equiv g)]

= \bar{f}_x \bar{g} + f_x \ast g

= f[x->g]

Functional substitution can be reduced to the application of the \(\exists\)-operator
Structural methods

- Functional methods limited by memory consumption of OBDD’s
- many circuits "similar", e.g., after simple technology mapping, buffer insertion, etc.
- basic idea: divide and conquer
  - partition circuit into sub-circuits by introducing cut-points
  - express functions of sub-circuits in terms of cut-point variables

Problems:
- how to find cut-points?
  - simple methods: equality by name, random simulation
- how to cope with false negatives?
Observation (Kühlmann DAC’97):
- in 80% of all circuit-pairs there 80% or more nodes that have equivalent nodes in the other circuit

Equivalence-proof based on circuit structure:
- conclude the equivalence of two gate-outputs from
  - pairwise equivalence of gate inputs
  - same gate-function

Example (Matsunaga DAC’96, modified)

1. Method:
   Prove \( v_1(a,b,c,d) \approx v_2(a,b,c,d) \)
   and \( w_1(a,b,c,d) \approx w_2(a,b,c,d) \)

2. Method:
   - Prove \( s_1(a,b,d) \approx s_2(a,b,d) \)
   and \( t_1(b,c) \approx t_2(b,c) \).
   If equivalent, \( s_1/s_2 \) and \( t_1/t_2 \) are cut-points
   - Prove \( v_1(c,s_1) \approx v_2(c,s_2) \)
   and \( w_1(s_1,t_1) \approx w_2(s_2,t_2) \)
Non-canonical graph representation of circuits (Kühlmann DAC’97) (similar representations are well-known from technology-mapping problems)

Non-canonical graph representation (Kühlmann DAC’97)
two nodes are equivalent iff the predecessors are equivalent (modulo negation)

\[ t_1 \text{ and } t_2 \text{ are equivalent} \]

building the (small !) OBDD’s for \( s_1 \) and \( s_2 \) we can prove that \( s_1 \) and \( s_2 \) are equivalent

development of OBDD’s is controlled by their size

by pure structural reasoning, we can show that \( v_1 \) and \( v_2 \) are equivalent
it remains to show that $w_1$ and $w_2$ are equivalent

expressing $w_1$ and $w_2$ in terms of cut-point variables $s_1/t_1$ and $s_2/t_2$, respectively we conclude that $w_1$ is not equivalent to $w_2$

Method 1: substitute functions in output functions

Substitute $s_1/t_1$ in $w_1$ and $s_2/t_2$ in $w_2$ and prove equivalence:

$w_1 = s_1 \oplus t_1 = abd \oplus bc$,

$w_2 = s_2 + t_2 = abd + bc$
Method 2: work on the exor of the outputs by substitution (Matsunaga DAC ’97) or case analysis (Kunz et al. DAC ’95)

Prove:
\[(w_1 \oplus w_2) = s_1^*t_1 = 0\]
by substitution:
\[adb*b*c = 0\]

Method 3: calculate characteristic function of image

Prove tautology:
\[\chi_i \Rightarrow (w_1 \equiv w_2) = 1\]
by image calculation:
\[(s_1 + t_1) \Rightarrow (w_1 = w_2)\]
\[(s_1 + t_1) \Rightarrow (s_1 + t_1) = 1\]
Satisfiability Checker

- SAT checker
  - rather than to demonstrate the tautology \( f = 1 \) positively, show that \( f = 0 \) leads to a contradiction
  - many modern SAT checker represent logical formulas as a conjunction of "triplets" of the form \( x = a \equiv b \) where \( a, b \) are literals
  - correspondence: \( x_1 = \bar{x}_2 \cdot a, x_2 = b \cdot \bar{c} \)

projection of SAT checking on circuit representation:

example: prove tautology \([a(b + c) \Rightarrow ab + ac] = 1\)
  - derive corresponding circuit with 2-input AND’s and inverters
  - try to produce a 0 at the output
  - propagate effect of value(s) until contradiction found
© HE99

- example: prove tautology \([a + c) \Rightarrow ab + ac] = 1\)

- another example: prove tautology \([a + c) \Leftarrow ab + ac] = 1\)
another example: prove tautology \[ a(b + c) \leq ab + ac \] = 1

1-saturation: case-split
- for all (input+intervenient) variables \( x \):
  - 0-saturation with \( x=1 \) and \( x=0 \)
  - if both lead to a contradiction: ✓
  - otherwise, record information common for both cases, proceed with next variable
    - breadth-first process

2-saturation
- 0-saturation with all combinations of 2 variables
- etc.
Current renaissance of SAT procedures, in particular:
- Stålmarch's procedure
  - patented algorithm (commercialized by Logikkonsult)
  - see tutorial Sheeran/Stålmarch at FMCAD'98
- Intelligent house-keeping of derived equalities in SAT checkers
- very large (10^5) #variables tractable
- application to various industrial problems (railway interlocking systems, engine managment units, ...)
- some similarity to "recursive learning" by Kunz/Pradhan
Different techniques are appropriate for different classes of circuits

Verification tools combine several techniques

- Kuehlmann/Krohm DAC’97 (simulation, OBDD’s, structural methods)
- Mukherjee et al. IWLS’97 (simulation, OBDD’s, structural methods, SAT-checker)
- Burch/Singhal ICCAD’98 (simulation, OBDD’s, structural methods, SAT-checker)