## 2. Verification by Equivalence Checking

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational Circuits Verification</td>
<td>2.2</td>
</tr>
<tr>
<td>Propositional Logic (Calculus)</td>
<td>2.5</td>
</tr>
<tr>
<td>Propositional Resolution</td>
<td>2.11</td>
</tr>
<tr>
<td>Stålmarck’s Procedure</td>
<td>2.19</td>
</tr>
<tr>
<td>Reduced Ordered Binary Decision Diagrams (ROBDDs)</td>
<td>2.23</td>
</tr>
<tr>
<td>Sequential Circuits Verification</td>
<td>2.55</td>
</tr>
<tr>
<td>Relational Representation of FSMs</td>
<td>2.56</td>
</tr>
<tr>
<td>Relational Product of FSMs</td>
<td>2.60</td>
</tr>
<tr>
<td>Reachability Analysis on FSMs</td>
<td>2.62</td>
</tr>
<tr>
<td>Equivalence Checking Tools</td>
<td>2.71</td>
</tr>
<tr>
<td>References</td>
<td>2.73</td>
</tr>
</tbody>
</table>
Combinational Circuits Verification

- Consist of an interconnection of logic gates — AND, OR, NOT, NAND, NOR, XOR, XNOR, and blocks implementing more complex logic (Boolean) functions.
- No logical loops, i.e., topologically there may be loops, but they are not sensitizable under any (valid) input combination, even such loops may be prohibited / not produced by automated analysis / synthesis tools

Goal

Given two Boolean netlists, check if the corresponding outputs of the two circuits are equal for all possible inputs
- Two circuits are equivalent iff the Boolean function representing the outputs of the networks are logically equivalent
- Identify equivalence points and implications between the two circuits to simplify equivalence checking
- Since a typical design proceeds by a series of local changes, in most cases there are many implications / equivalent subcircuits in the two circuits to be compared
- Various tautology/satisfiability checking algorithms based on heuristics (problem is NP-complete, but many work well on “real” applications ...)
- In this course we consider three main combinational equivalence checking methods:
  - Propositional resolution method (tautology/satisfiability checking)
  - Stålmarck’s method (recent patented algorithm, very efficient and popular)
  - ROBDD-based method (Boolean function converted into ROBDD’s representation)
Combinational Equivalence Checking

Explicit Proof

- Propositional resolution
- Stålmarck’s procedure
- ROBDDs

\((f_1 = f_2) = T\)
Combinational Equivalence Checking (con’t)

Implicit Proof

- ROBDDs
Propositional Logic (Calculus)

Syntax

- $P, Q, R,...$ — propositional symbols (atomic propositions)
- $t$: true; $f$: false — constants
- $\neg P$: not $P$
- $P \land Q$: $P$ and $Q$
- $P \lor Q$: $P$ or $Q$
- $P \rightarrow Q$: if $P$ then $Q$ (proposition equivalent to $\neg P \lor Q$)
- $P \leftrightarrow Q$: $P$ if and only if $Q$, i.e., $P$ equivalent to $Q$

(proposition equivalent to $(P \land Q) \lor (\neg P \land \neg Q)$)

Semantics

Given through the Truth Table:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$Q$</td>
<td>$\neg P$</td>
<td>$P \land Q$</td>
<td>$P \lor Q$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$f$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$t$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
<td>$t$</td>
</tr>
<tr>
<td>$f$</td>
<td>$t$</td>
<td>$t$</td>
<td>$f$</td>
<td>$t$</td>
</tr>
<tr>
<td>$f$</td>
<td>$f$</td>
<td>$t$</td>
<td>$f$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

An interpretation is a function from the propositional symbols to $\{t, f\}$
Propositional Logic (cont’d)

• Formula F is **satisfiable** (consistent) iff it is **true** under **at least one** interpretation
• Formula F is **unsatisfiable** (inconsistent) iff it is **false** under **all** interpretations
• Formula F is **valid** iff it is **true** (consistent) under **all** interpretations
• Interpretation I satisfies a formula F (I is a **model** of F) iff F is true under I. Notation: I \models F

• **Theorem:** A formula F is valid (a **tautology**) iff \( \neg F \) is unsatisfiable. Notation: \( \models F \)
• The relationship between F to \( \neg F \) can be visualized by “mirror principle”:

<table>
<thead>
<tr>
<th>Valid formulas</th>
<th>Satisfiable, but non-valid formulas</th>
<th>Unsatisfiable formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>G ↦ F</td>
<td>( F ) ↦ ↦ ( \neg F )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \neg G ) ↦ ( \neg F )</td>
<td></td>
</tr>
</tbody>
</table>

• To determine if F is satisfiable or valid, test finite number \( (2^n) \) of interpretations of the \( n \) atomic propositions occurring in F
  ... but it is an exponential method... **satisfiability is an NP-complete problem**
Propositional Logic (cont’d)

Proofs

• A proof of a proposition is derived using axioms, theorems, and inference rules (an inference rule permits deducing conclusions based on the truth of certain premises)

• A logic formula F is deducible from the set S of statements if there is a finite proof of F starting from elements of S. Notation: S ⊢ F

Example: A simple proof system

• Axioms: K: A → (B → A)
  
  S: (A → (B → C)) → ((A → B) → (A → C))

  DN: ¬¬A → A

• Inference rule (Modus Ponens): {A → B, A} ⊢ B

• A proof of A → A

  (1) ⊢ (A → ((D → A) → A)) → ((A → (D → A)) → (A → A)) by S ([B\D→A], [C\A])

  (2) ⊢ A → ((D → A) → A) by K ([B\D→A])

  (3) ⊢ (A → (D → A)) → (A → A) by MP, (1), (2)

  (4) ⊢ A → (D → A) by K

  (5) ⊢ A → A by MP, (3), (4).
Propositional Logic (cont’d)

Relation between syntax and semantics

- Truth tables provide a means of deciding truth
- Propositional logic is:
  - **complete**: everything that is true may be proven, i.e., if $S \models A$ then $S \models A$
  - **consistent (sound)**: nothing that is false may be proven. i.e., if $S \models A$ then $S \vdash A$
  - **decidable**: there is an algorithm for deciding the truth of any proposition, i.e., test a finite (exponential) number of truth assignments
False Negative & False Positive

Let P be a proposition (a property) and A a verification method (algorithm).

- **False Negative**: (similar to incompleteness)

  A(P) reports true $\Rightarrow \forall$ interpretation $\psi$, $\psi(P) = true$
  
  A(P) reports false $\Rightarrow \neg(\forall$ interpretation $\psi$, $\psi(P) = true)$ ! (\exists \psi, \psi(P) = false)

- **False Positive**: (similar to inconsistency, unsoundness)

  A(P) reports false $\Rightarrow \forall$ interpretation $\psi$, $\psi(P) = false$
  
  A(P) reports true $\Rightarrow \neg(\forall$ interpretation $\psi$, $\psi(P) = false)$ ! (\exists \psi, \psi(P) = true)
Combinational Equivalence Checking

- Determine if two expressions $f_1$ and $f_2$ denote the same truth table

- Application: Determine if two combinational logic circuit designs $C_1$ and $C_2$ implement the same truth table (logic (Boolean) function)
  - Extract representation of logic expressions $f_1$ and $f_2$
  - Verify if
    
    $(f_1 \leftrightarrow f_2)$ is a valid formula, i.e., $\neg(f_1 \leftrightarrow f_2)$ is unsatisfiable using satisfiability algorithms (Propositional Resolution methods), or

    $(f_1 \rightarrow f_2)$ and $(f_2 \rightarrow f_1)$ hold (where $f_1$ and $f_2$ are transformed to implication form using Stålmardck’s procedure), or

    $f_1$ and $f_2$ have the same canonical form using, e.g., Reduced Binary Decision Diagrams
Propositional Resolution

- A **Literal** $L$ is an atomic proposition $A$ or its negation $\neg A$
- A **Clause** $C$ is a finite set of disjunctive literals ($C = L_1 \lor L_2 \lor L_3 \lor ...$)
  
  $C$ is true iff one of its elements is true. The empty clause $\Box$ is always false.

Let $A_1, A_2, ...$ be atomic propositions and $L_{i,j}$ literals

- **Conjunctive Normal Form** (CNF): a conjunction of disjunctions of literals
  \[
  F = (\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} L_{i,j})), \text{ where } L_{i,j} \in \{A_1, A_2, ...\} \cup \{\neg A_1, \neg A_2, ...\}
  \]

- **Disjunctive Normal Form** (DNF): a disjunction of conjunctions of literals
  \[
  F = (\bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} L_{i,j})), \text{ where } L_{i,j} \in \{A_1, A_2, ...\} \cup \{\neg A_1, \neg A_2, ...\}
  \]

Each $L_{i,j} \in \{A_1, A_2, ...\} \cup \{\neg A_1, \neg A_2, ...\}$ appears in each disjunct (conjunct) **at most once**!

**Theorem**: For every logic formula $F$, there is an equivalent CNF and an equivalent DNF

- **Canonical Conjunctive Form** (CCF): CNF in which each $L$ appears exactly once
- **Canonical Disjunctive Form** (DCF): DNF in which each $L$ appears exactly once
Propositional Resolution (cont’d)

- **Resolution** is a proof method underlying some automatic theorem provers based on simple syntactic transformation and *refutation*.
- **Refutation** is a procedure to show that a given formula is unsatisfiable

**Resolution procedure:**
- To prove $F$, we translate $\neg F$ into a set of clauses, each a disjunction of atomic formulae or their negations.
- Each resolution step takes two clauses and yields a new one.
- The method succeeds if it produces the empty clause (a contradiction), thus refuting $\neg F$. 
Propositional Resolution (cont’d)

• Let \( F = (L_{1,1} \lor \ldots \lor L_{1,n_1}) \land \ldots \land (L_{k,1} \lor \ldots \lor L_{k,n_k}) \) where literals \( L_{i,j} \in \{A_1,A_2,\ldots\} \cup \{\neg A_1,\neg A_2,\ldots\} \)

\( F \) can be viewed as a set of clauses: \( F = \{\{L_{1,1},\ldots, L_{1,n_1}\},\ldots, \{L_{k,1},\ldots, L_{k,n_k}\}\} \), where
  - Comma separating two literals within a clause corresponds to \( \lor \)
  - Comma separating two clauses corresponds to \( \land \)

• Let \( L \) be a literal in clause \( C_1 \) (\( L \in C_1 \)) and its complement \( \overline{L} \) in clause \( C_2 \) (\( \overline{L} \in C_2 \)),

Clause \( R \) is a **resolvent** of \( C_1 \) and \( C_2 \) if: \( R = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\}) \)

• Example: \( F = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, v\}, \{\neg s\}, \{s, \neg v\}\} \).

![Diagram of Propositional Resolution](attachment:image.png)
Propositional Resolution (cont’d)

- A \textit{(resolution) deduction} of C from F is a finite sequence $C_1, C_2, \ldots, C_n$ of clauses such that each $C_i$ is either in F or a resolvent of $C_j, C_k$, $(j, k < i)$

- $\text{Res}(F) = F \cup R$ where R is a resolvent of two clauses in F

\textbf{Lemma}. F and $F \cup R$ are equivalent

- Define
  
  $\text{Res}^0(F) = F,$
  
  $\text{Res}^{n+1}(F) = \text{Res}(\text{Res}^n(F)),$ $n \geq 0$

- Let $\text{Res}^*(F) = \bigcup_{n \geq 0} \text{Res}^n(F)$

\textbf{Theorem}. F is unsatisfiable iff $\Box \in \text{Res}^*(F)$

- Algorithm: to decide satisfiability of formula F in CNF (clause set):

  \begin{verbatim}
  repeat
  G:=F;
  F:=\text{Res}(F)
  until ((\Box \in F) or (F = G));
  if \Box \in F then “F is unsatisfiable” else “F is satisfiable”.
  \end{verbatim}
Propositional Resolution (cont’d)

Summary of basic idea:

Goal: \( G \) in DNF is valid?

\(-G\) is unsatisfiable

\( F = \neg G \)

in CNF: \( F = (L_{1,1} \lor \ldots \lor L_{1,n_1}) \land \ldots \land (L_{k,1} \lor \ldots \lor L_{k,n_k}) \)

\( F = \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{k,1}, \ldots, L_{k,n_k}\}\} \)

Resolution Refutation procedure

\( F = \{\square\} \) (contradiction)
Propositional Resolution - Example

Two circuits C1 and C2

Propositional Resolution

C1: out1 = a ∨ b

C2: out2 = (¬ a ∧ b) ∨ (a ∧ a)

(Mux: out2 = (¬ s ∧ b) ∨ (s ∧ a))

G = (out1 ⇔ out2)
\[ G = (\text{out 1} \land \text{out 2}) \lor (\neg\text{out 1} \land \neg\text{out 2}) \quad \text{(DNF)} \]

\[ = \text{true?} \]

\[ F = \neg G = \neg ((\text{out 1} \land \text{out 2}) \lor (\neg\text{out 1} \land \neg\text{out 2})) \]

\[ = \text{False? (unsatisfiable!)} \]

**CNF**

\[ F = (\neg\text{out 1} \lor \neg\text{out 2}) \land (\text{out 1} \lor \text{out 2}) \]

\[ = (\neg(a \lor b) \lor \neg [(\neg a \land b) \lor (a \land a)]) \land ((a \lor b) \lor [(\neg a \land b) \lor (a \land a)]) \]

\[ = \ldots. \]

\[ = (\neg a) \land (\neg b) \land (a \lor b) \]

**Literals:** \{\{\neg a\}, \{\neg b\}, \{a, b\}\}
Theorem Proving

\[ \text{out1} = (\neg s \land b) \lor (s \land a) \]
\[ = (\neg a \land b) \lor (a \land a) \]
\[ = (\neg a \land b) \lor a \]
\[ = (\neg a \lor a) \land (b \lor a) \]
\[ = 1 \land (b \lor a) \]
\[ = b \lor a = a \lor b \]
\[ \Rightarrow \text{out2} = \text{out1} \]
Stålmarck’s Procedure

• Transform propositional formula $G$ (in linear time) in a nested implication form, e.g.: $G = (p \rightarrow (q \rightarrow r)) \rightarrow s$

• $G$ is now represented using a set of triplets $\{b_i, x, y\}$, meaning “$b_i \leftrightarrow (x \rightarrow y)$”, e.g.: $(p \rightarrow (q \rightarrow r)) \rightarrow s$ becomes $\{(b_1, q, r), (b_2, p, b_1), (b_3, b_2, s)\}$; $G = b_3$

• To prove a formula valid, assume that it is false and try to find a contradiction (use 0 for false and 1 for true, as in switching (Boolean) algebra)

• Derivation rules: (a/b means “replace a by b”)
  - $r_1 (0, y, z) \Rightarrow y/1, z/0$ meaning $false \leftrightarrow (y \rightarrow z)$ implies $y = true$ and $z = false$
  - $r_2 (x, y, 1) \Rightarrow x/1$ meaning $x \leftrightarrow (y \rightarrow true)$ implies $x = true$
  - $r_3 (x, 0, z) \Rightarrow x/1$ meaning $x \leftrightarrow (false \rightarrow z)$ implies $x = true$
  - $r_4 (x, 1, z) \Rightarrow x/z$ meaning $x \leftrightarrow (true \rightarrow z)$ implies $x = z$
  - $r_5 (x, y, 0) \Rightarrow x/\neg y$ meaning $x \leftrightarrow (y \rightarrow 0)$ implies $x = \neg y$
  - $r_6 (x, x, z) \Rightarrow x/1, z/1$ meaning $x \leftrightarrow (x \rightarrow z)$ implies $x = true$ and $z = true$
  - $r_7 (x, y, y) \Rightarrow x/1$ meaning $x \leftrightarrow (y \rightarrow y)$ implies $x = true$

Example: $G = (p \rightarrow (q \rightarrow p)) : \{(b_1, q, p), (b_2, p, b_1)\}$, assume $G = b_2 = 0$, i.e., $(0, p, b_1)$
By $r_1 : p = 1$ and $b_1 = 0$, substitute for $b_1$ and get $(0, q, 1)$ (which is a terminal triplet)
Again by $r_1$ this is a contradiction since $1/0$ is derived for $z$ in $r_1$, hence $b_2 = G = 1$ (true)
Stålmarck’s Procedure (cont’d)

- Not all formulas can be proved with these rules, need a form of branching: **Dilemma rule**
  
  $T = \{D_i, i = 1, 2, \text{ are derivations, results } U[S_1] \text{ and } V[S_2], \text{ conclusion } T[S]\}$

  $\begin{array}{cccc}
  & T & & T \\
  T[x/1] & T[x/0] & & & \\
  D_1 & D_2 & & & \\
  U[S_1] & V[S_2] & & & \\
  T[S] & & & &
  \end{array}$

Assume $x = 0$ derive a result, then assume $x = 1$ and also derive a result.

- If either derivation gives a contradiction, the result is the other derivation
- If both are contradictions, then $T$ contains a contradiction
- Otherwise return the intersection of the result of the two derivations, since any information gained from $x = 0$ and $x = 1$ must be independent of that value

Example: $T = \{(1, \neg p, p), (1, p, \neg p)\}$ cannot be resolved using $r1 - r7$

$T[p/1] = \{(1, 0, 1), (1, 1, 0)\}$ where $(1, 1, 0)$ is a contradiction

$T[p/0] = \{(1, 1, 0), (1, 0, 1)\}$ where $(1, 1, 0)$ is again a contradiction

Hence $T[S]$ results in a contradiction.
Stålmarck’s Procedure (cont’d)

Transformation from and-or-not logic to implication form:

**not:** \( G = \neg A \iff A \to 0 \iff \{(x, A, 0)\} \) , \( G = x \)

**or:** \( G = A \lor B \iff \neg A \to B \iff \{(x, y, B), (y, A, 0)\} \) , \( G = x \)

**and:** \( G = A \land B \iff \neg(A \to \neg B) \iff \{(x, y, 0), (y, A, z), (z, B, 0)\} \) , \( G = x \)

Example of equivalence checking:

\[
\begin{array}{c}
\text{C1} = \{(y, e, x), (e, b, 0), (x, f, 0), \\
(f, a, g), (g, b, 0)\}
\end{array}
\]

\[
\begin{array}{c}
\text{C2} = \{(t, h, s), (h, b, 0), (s, u, 0), (u, r, v), (v, c, 0), \\
(r, w, 0), (w, a, p), (p, b, 0)\}
\end{array}
\]

Check \( y \to t \) and \( t \to y \)

\( y \to t \): Form \( C1 \cup C2 \cup \{(0, y, t)\} \) which by r1 yields \([y/1, t/0]\) and after substitution

\[
\{(1, e, x), (e, b, 0), (x, f, 0), (f, a, g), (g, b, 0), (0, h, s), (h, b, 0), (s, u, 0), (u, r, v), (v, c, 0), \\
(r, w, 0), (w, a, p), (p, b, 0)\}\]

giving by r1 again \([h/1, s/0]\) and...
Example of equivalence checking (cont’d):

\{(1, e, x), (e, b, 0), (x, f, 0), (f, a, g), (g, b, 0), (1, b, 0), (0, u, 0), (u, r, v), (v, c, 0), (r, w, 0), (w, a, p), (p, b, 0)\} apply r1 and r5 and get [u/1, \neg b, x/\neg f, g/\neg b, v/\neg c, r/\neg w, p/\neg b] which yields

\{(1, \neg b, \neg f), (f, a, \neg b), (1, b, 0), (1, \neg w, \neg c), (w, a, \neg b)\}

Application of Dilemma rule to, say, b yields:

\(b = 0: \{(1, 1, \neg f), (f, a, 1), (1, 0, 0), (1, \neg w, \neg c), (w, a, 1)\}\) yields \[f/1, w/1\] by r2, thus

\{(1,1, 0), (1, a, 1), (1, 0, \neg c), (1, a, 1)\} i.e., \((1,1, 0)\) is a contradiction

\(b = 1: \{(1, 0, \neg f), (f, a, 0), (1, 1, 0), (1, \neg w, \neg c), (w, a, 0)\}\) a contradiction again

Conclusion: \(y \rightarrow t\) holds.
Similarly for \(t \rightarrow y\)

*The two circuits are equivalent.*
Binary Decision Diagrams (BDDs)

Classical representation of logic functions: Truth Table, Karnaugh Maps, Sum-of-Products, critical complexes, etc.

- Critical drawbacks:
  - May not be a canonical form or is too large (exponential) for “useful” functions,
    ⇒ Equivalence and tautology checking is hard
  - Operations like complementation may yield a representation of exponential size

Reduced Ordered Binary Decision Diagrams (ROBDDs)

- A canonical form for Boolean functions
- Often substantially more compact than traditional normal forms
- Can be efficiently manipulated
- Introduced mainly by R. E. Bryant (1986).
- Various extensions exist that can be adapted to the situation at hand (e.g., the type of circuit to be verified)
Binary Decision Trees

- A Binary decision Tree (BDT) is a *rooted, directed graph* with *terminal and nonterminal vertices*.
- Each nonterminal vertex $v$ is labeled by a variable $\text{var}(v)$ and has two successors:
  - $\text{low}(v)$ corresponds to the case where the variable $v$ is assigned 0
  - $\text{high}(v)$ corresponds to the case where the variable $v$ is assigned 1
- Each terminal vertex $v$ is labeled by $\text{value}(v) \in \{0, 1\}$
- **Example**: BDT for a two-bit comparator, $f(a_1,a_2,b_1,b_2) = (a_1 \Leftrightarrow b_1) \land (a_2 \Leftrightarrow b_2)$
Binary Decision Trees (cont’d)

- We can decide if a truth assignment \( x = (x_1, \ldots, x_n) \) satisfies a formula in BDT in linear time in the number of variables by traversing the tree from the root to a terminal vertex:
  - If \( var(v) \in x \) is 0, the next vertex on the path is \( low(v) \)
  - If \( var(v) \in x \) is 1, the next vertex on the path is \( high(v) \)
  - If \( v \) is a terminal vertex then \( f(x) = f_v(x_1, \ldots, x_n) = value(v) \)
  - If \( v \) is a nonterminal vertex with \( var(v) = x_i \), then the structure of the tree is obtained by Shanon’s expansion
    \[
    f_v(x_1, \ldots, x_n) = [\neg x_i \land f_{low(v)}(x_1, \ldots, x_n)] \lor [x_i \land f_{high(v)}(x_1, \ldots, x_n)]
    \]
- For the comparator, \( (a_1 \leftarrow 1, a_2 \leftarrow 0, b_1 \leftarrow 1, b_2 \leftarrow 1) \) leads to a terminal vertex labeled by 0, i.e., \( f(1, 0, 1, 1) = 0 \)
- Binary decision trees are redundant:
  - In the comparator, there are 6 subtrees with roots labeled by \( b_2 \), but not all are distinct
- Merge isomorphic subtrees:
  - Results in a directed acyclic graph (DAG), a binary decision diagram (BDD)
Reduced Ordered BDD

Canonical Form property

- A *canonical* representation for Boolean functions is desirable:
  - two Boolean functions are logically equivalent iff they have isomorphic representations
- This simplifies checking equivalence of two formulas and deciding if a formula is satisfiable
- Two BDDs are **isomorphic** if there exists a bijection $h$ between the graphs such that
  - Terminals are mapped to terminals and nonterminals are mapped to nonterminals
  - For every terminal vertex $v$, $\text{value}(v) = \text{value}(h(v))$, and
  - For every nonterminal vertex $v$:
    \[
    \text{var}(v) = \text{var}(h(v)), \quad h(\text{low}(v)) = \text{low}(h(v)), \quad \text{and} \quad h(\text{high}(v)) = \text{high}(h(v))
    \]
- **Bryant (1986) showed that BDDs are a canonical representation** for Boolean functions under two restrictions:
  1. the variables appear **in the same order along each path** from the root to a terminal
  2. there are no isomorphic subtrees or redundant vertices

$\Rightarrow$ Reduced Ordered Binary Decision Diagrams (ROBDDs)
Canonical Form Property

• Requirement (1): Impose total order “<” on the variables in the formula:
  if vertex \( u \) has a nonterminal successor \( v \), then \( \text{var}(u) < \text{var}(v) \)

• Requirement (2): repeatedly apply three transformation rules (or implicitly in operations such as disjunction or conjunction)

1. Remove duplicate terminals: eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one
2. **Remove duplicate nonterminals**: if nonterminals $u$ and $v$ have $\text{var}(u) = \text{var}(v)$, $\text{low}(u) = \text{low}(v)$ and $\text{high}(u) = \text{high}(v)$, eliminate one of the two vertices and redirect all incoming arcs to the other vertex.

![Diagram](image1)

3. **Remove redundant tests**: if nonterminal vertex $v$ has $\text{low}(v) = \text{high}(v)$, eliminate $v$ and redirect all incoming arcs to $\text{low}(v)$.

![Diagram](image2)
Creating the ROBDD for \((x \oplus y \oplus z)\)
Canonical Form Property (cont’d)

- A canonical form is obtained by applying the transformation rules until no further application is possible
- Bryant showed how this can be done by a procedure called **Reduce** in linear time
- Applications:
  - *checking equivalence*: verify isomorphism between ROBDDs
  - *non-satisfiability*: verify if ROBDD has only one terminal node, labeled by 0
  - *tautology*: verify if ROBDD has only one terminal node, labeled by 1

**Example:**
ROBDD of 2-bit Comparator with variable order $a_1 < b_1 < a_2 < b_2$: 

![Diagram of 2-bit Comparator ROBDD]
ROBDD Examples

OR

\[ \text{out} = f(a, b) = a \lor b \]

\[ \begin{array}{c|c|c|c|c|c} a & b & \text{out} \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ \end{array} \]

BDD

ROBDD
ROBDD Examples (con’t)

AND

\[
\text{out} = f(a,b) = a \land b
\]

BDD

ROBDD
ROBDD Examples (con’t)

\[
\text{XOR} \\
a \\
b \\
\text{out} = f(a,b) = a \oplus b
\]
ROBDD Examples (con’t)

\[
\text{out} = f(a, b) = \neg (a \land b)
\]

**NAND**

\[
\begin{array}{c}
a \\
b
\end{array} \quad \text{out} = f(a, b) = \neg (a \land b)
\]

**BDD**

\[
\begin{array}{c|c|c}
a & b & \text{out} \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

**ROBDD**

\[
\begin{array}{c}
a \\
b
\end{array}
\]

**Reduction**

\[
\begin{array}{c|c|c}
a & b & \text{out} \\
0 & - & 1 \\
- & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Variable Ordering Problem

- The size of an ROBDD depends critically on the variable order.
- For order \(a_1 < a_2 < b_1 < b_2\), the comparator ROBDD becomes:

![Diagram of a comparator ROBDD]

- For an \(n\)-bit comparator:
  - \(a_1 < b_1 < ... < a_n < b_n\) gives \(3n+2\) vertices (linear complexity)
  - \(a_1 < ... < a_n < b_1 < ... < b_n\), gives \(3 	imes 2^n - 1\) vertices (exponential complexity!)
Variable Ordering Problem - Example

$$(x_1 \oplus y_1) \lor (x_2 \oplus y_2) \lor (x_3 \oplus y_3)$$
Variable Ordering Problem (cont’d)

- The problem of finding the *optimal* variable order is NP-complete
- Some Boolean functions have exponential size ROBDDs for any order (e.g., multiplier)

**Heuristics for Variable Ordering**
- Heuristics developed for finding a *good* variable order (if it exists)
- Intuition for these heuristics comes from the observation that ROBDDs tend to be smaller when related variables are close together in the order (e.g., ripple-carry adder)
- Variables appearing in a subcircuit are related: they determine the subcircuit’s output

  ⇒ should usually be close together in the order

**Dynamic Variable Ordering**
- Useful if no obvious static ordering heuristic applies
- During verification operations (e.g., reachability analysis) functions change, hence initial order is not good later on
- Good ROBDD packages periodically internally reorder variables to reduce ROBDD size
- Basic approach based on neighboring variable exchange ... < a < b < ... ⇒ ...< b < a < ...
  Among a number of trials the best is taken, and the exchange is repeated
Logic Operations on ROBDDs

- Residual function (cofactor): \( b \in \{0, 1\} \)

\[ f \Big|_{x_i \leftarrow b} (x_1, ..., x_n) = f(x_1, ..., x_{i-1}, b, x_{i+1}, ..., x_n) \]

- ROBDD of \( f \Big|_{x_i \leftarrow b} \) computed by a depth-first traversal of the ROBDD of \( f \):
  - For any vertex \( v \) which has a pointer to a vertex \( w \) such that \( \text{var}(w) = x_i \), replace the pointer by \( \text{low}(w) \) if \( b \) is 0 and by \( \text{high}(w) \) if \( b \) is 1.
  - If not in canonical form, apply Reduce to obtain ROBDD of \( f \Big|_{x_i \leftarrow b} \).

- All 16 two-argument logic operations on Boolean function implemented efficiently on ROBDDs in linear time in the size of the argument ROBDDs.
Logic Operations on ROBDDs (cont’d)

- Based on Shannon’s expansion

\[ f = \left[ -x \land f \right|_{x=0} \lor \left[ x \land f \right|_{x=1} \right] \]

- Bryant (1986) gave a uniform algorithm, *Apply*, for computing all 16 operations:
  - \( f \ast f' \): an arbitrary logic operation on Boolean functions \( f \) and \( f' \)
  - \( v \) and \( v' \): the roots of the ROBDDs for \( f \) and \( f' \), \( x = \text{var}(v) \) and \( x' = \text{var}(v') \)

- Consider several cases depending on \( v \) and \( v' \)
  1. \( v \) and \( v' \) are both terminal vertices: \( f \ast f' = \text{value}(v) \ast \text{value}(v') \)
  2. \( x = x' \): use Shannon’s expansion

\[ f \ast f' = \left[ -x \land (f \left| x=0 \right. \ast f' \left| x=0 \right.) \right] \lor \left[ x \land (f \left| x=1 \right. \ast f' \left| x=1 \right.) \right] \]

- to break the problem into two subproblems, each is solved recursively
  - The root is \( v \) with \( \text{var}(v) = x \)
  - \( \text{Low}(v) \) is \( (f \left| x=0 \right. \ast f' \left| x=0 \right.) \)
  - \( \text{High}(v) \) is \( (f \left| x=1 \right. \ast f' \left| x=1 \right.) \)
Logic Operations on ROBDDs (cont’d)

(3) $x < x'$: $f' \mid _{x \leftarrow 0} = f' \mid _{x \leftarrow 1} = f'$ since $f'$ does not depend on $x$

In this case the Shannon's expansion simplifies to

$$f \cdot f' = \left[ -x \land (f \mid _{x \leftarrow 0} \cdot f') \right] \lor \left[ x \land (f \mid _{x \leftarrow 1} \cdot f') \right],$$

similar to (2) and compute subproblems recursively,

(4) $x' < x$: similar to the case above

**Improvement using the if-then-else (ITE) operator:**

$$\text{ITE}(F, G, H) = F \cdot G + F' \cdot H$$

where $F$, $G$ and $H$ are functions

Recursive algorithm based on the following, $v$ is the top variable (lowest index):

$$\text{ITE}(F, G, H) = v.(F \cdot G + F' \cdot H) + v'.(F \cdot G + F' \cdot H)$$

$$= v.(F_v \cdot G_v + F'_v \cdot H_v) + v'.(F_v \cdot G_v + F'_v \cdot H_v')$$

$$= (v, \text{ITE}(F_v, G_v, H_v), \text{ITE}(F'_v, G'_v, H'_v))$$

With terminal cases being: $F = \text{ITE}(1, F, G) = \text{ITE}(0, G, F) = \text{ITE}(F, 1, 0) = \text{ITE}(G, F, F)$

we define

$$\text{NOT}(F) = \text{ITE}(F, 0, 1)$$

$$\text{AND}(F, G) = \text{ITE}(F, G, 0)$$

$$\text{OR}(F, G) = \text{ITE}(F, 1, G)$$

$$\text{XOR}(F, G) = \text{ITE}(F, \neg G, G)$$

$$\text{LEQ}(F, G) = \text{ITE}(F, G, 1)$$

etc.
Logic Operations on ROBDDs (cont’d)

- By using **dynamic programming**, it is possible to make the ITE algorithm polynomial:
  1. The result must be reduced to ensure that it is in canonical form;
     - record constructed nodes (*unique table*);
     - before creating a new node, check if it already exists in this *unique* hash table
  2. Record all previously computed functions in a hash table (*computed table*);
     - must be implemented efficiently as it may grow very quickly in size;
     - before computing any function, check table for solution already obtained

- **Complement edges** can reduce the size of an ROBDD by a factor of 2
  - Only one terminal node is labeled 1
  - Edges have an attribute (dot) to indicate if they are inverting or not
  - To maintain canonicity, a dot can appear only on *low(v)* edges
    - Complementation achieved in O(1) time by placing a dot on the function edge
    - F and F’ can share entry in *computed table*
    - Adaptation of ITE easy

- Test for $F \leq G$ can be computed by a specialized ITE_CONSTANT algorithm

![Comparator Diagram](image_comparator.png)
BDD Operators - Example

Task: compute ROBDD for \( f(a,b) \)

1) \( f = x \land y = (a \lor b) \land (a \land b) \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

order a,b.

![Diagram](image_url)
2) \( f = x \land y \)

\[
\text{BDD}_f = \text{“BDD}_x \land \text{BDD}_y”
\]

\( = \text{Conj (BDD}_x, \text{BDD}_y) \)

\[
\begin{array}{c}
\text{BDD}_x \\
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\end{array}
\end{array}
\begin{array}{c}
\text{BDD}_y \\
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\]

\( x \land 0 = 0 \)

\( x \land 1 = x \)

\[
\begin{array}{c}
\text{a = 1:} \quad 1 \land b = b \\
\text{a = 0:} \quad b \land 0 = 0 \\
\text{b = 1:} \quad 1 \land 1 = 1 \\
\text{b = 0:} \quad 0 \land 0 = 0
\end{array}
\]

2.43 (of 73)
Other Decision Diagrams

- Multiterminal BDD (MTBDD): Pseudo-Boolean functions \( B^n \rightarrow N \), terminal nodes are integers
- Binary Moment Diagrams (BMD): for representing and verifying arithmetic operations, word-level representation
- Ordered Kronecker Functional BDDs (OKFBDD): Based on XOR operations and OBDD
- Free BDDs (FBDD): Different variable order along different paths in the graph
- Zero suppressed BDDs (ZBDD)
- Combination of various forms of DDs integrated in DD software packages: Drechsler et al (U. Freiburg, Germany), Clarke et al (Carnegie Mellon U., USA)
- Extension to represent systems of linear and Boolean constraints (DTU)
- Multiway Decision Diagrams (MDG): Representation for a subset of equational first-order logic for modeling state machines with abstract and concrete data (U. of Montreal)

Well known ROBDD packages:
- CMU (as used in SMV from Carnegie Mellon U.)
- CUDD, U. of Colorado at Boulder (as used in VIS from UC at Berkeley)
- Industrial packages: Intel, Lucent, Cadence, Synopsys, Bull Systems, etc.
Applications of ROBDDs

ROBDD:
- Construction DD from circuit description:
  - Depth-first vs. breadth-first construction (keep only few levels in memory, rest on disk; problem with dynamic reordering)
  - Partitioning of Boolean space, each partition represented by a separate graph
  - Bottom-up vs. top-down, introducing decomposition points
- Internal correspondences in the two circuits — equivalent functions, or complex relations

ATPG-based:
- Combine circuits with an XOR gate on the outputs, show inexistence of test for a fault s-a-0 on the output (i.e., the output would have to be driven to 1 meaning that there is a difference in the two circuits)
- Use ATPG and learning to determine equivalent circuit nodes

Fast random simulation:
- Detect quickly easy differences

Real tools:
- Use a combination of techniques, fast and less powerful first, slow but exact later
Combinational Equivalence Checking - Example

Two circuits $C_1$ and $C_2$

$C_1$: \( a \lor b \) \hfill $C_2$: if \( a \) then \( a \) else \( b \) \hfill MUX: if \( c \) then \( a \) else \( b \)

\[ a \quad b \rightarrow \quad \text{out 1} \]
\[ a \quad b \rightarrow \quad \text{out 2} \]

\[ \text{isomorph} \]
Combinational Equivalence Checking – Multiplexor Example

**Specification:** if \( c = 1 \) then \( \text{out} = a \)
else \( \text{out} = b \)

Build ROBBD for Spec:

<table>
<thead>
<tr>
<th>( c )</th>
<th>( a )</th>
<th>( b )</th>
<th>( \text{out} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ROBDD1:
order: \( c, a, b \)

Implementation:

\[
\text{out} = (c \land a) \lor (\neg c \land b)
\]
Mutiplexor Example (con’t)

Build ROBDD for Imp:

\[ x: c \land a \]

\[ y: \neg c \land b \]

ROBDD2
order: c, a, b

isomorph to ROBDD1!
Multiplexor Example (con’t)

Alternative way to build ROBDD2:

\[
\text{out} = (c \land a) \lor (\neg c \land b)
\]

order: c, a, b

BBD

```
+---+---+---+
|   |   |   |
| c | a | b |
+---+---+---+
|   |   |   |
| 0 | 1 | 0 |
+---+---+---+
|   |   |   |
| 0 | 0 | 0 |
+---+---+---+
|   |   |   |
| 0 | 1 | 1 |
+---+---+---+
```

ROBBD

```
+---+---+---+
|   |   |   |
| c | a | b |
+---+---+---+
|   |   |   |
| 0 | 1 | 0 |
+---+---+---+
|   |   |   |
| 0 | 0 | 0 |
+---+---+---+
|   |   |   |
| 0 | 1 | 1 |
+---+---+---+
```

isomorph to ROBDD1
Comparator Example

Spec: \( f(\hat{a}, \hat{b}) = 1 \) if \( \hat{a} = \hat{b} \)

Refinement: \( f(a_1, a_2, b_1, b_2) = 1 \) if \( (a_1 = b_1) \land (a_2 = b_2) \)

\( \hat{a} = a_1a_2 \)

\( \hat{b} = b_1b_2 \)

Implicit:
Comparator Example (cont’d)

\[ z = (x \land y) \lor (\neg x \land \neg y) \]

\[
\begin{align*}
T_1 & \quad \quad & T_4 \\
T_2 & \quad \quad & T_3
\end{align*}
\]

\[
f = [(a_1 \land b_1) \lor (\neg a_1 \land \neg b_1)] \land \\
[(a_2 \land b_2) \lor (\neg a_2 \land \neg b_2)]
\]
Comparator Example (cont’d)

\[
\begin{align*}
T1: & \quad \text{disj: } T1, T2, T12 \\
T2: & \quad 0 \quad a1 \quad 1 \quad b1 \quad 0 \quad 1 \\
T3: & \quad a2 \quad 0 \quad b2 \quad 0 \quad 1 \\
T4: & \quad 0 \quad a2 \quad 1 \quad b2 \quad 0 \\
\end{align*}
\]
Comparator Example (cont’d)

conj. \( T_{12}, T_{34}, \) a₁, a₂

\( b_1, b_2 \) independent

order: a₁, b₁, a₂, b₂

Isomorph to the spec
Equivalence Checking in Practice

- Usually, combinational circuits implement arithmetic and logic operations, and next-state and output functions of finite-state machines (sequential circuits)
- Verifying the behavior of the gate-level implementation against the RTL design of digital systems can often be reduced to verifying the combinational circuits
  - Equivalence comparison between the next-state and output functions (combinational circuits)
  - Requires that both have the same state space (and of course inputs and outputs), knowing the mapping between states helps...
  - Can also be used to verify gate-level implementation against gate-level model extracted from layout
  - This kind of verification is useful for confirming the correctness of manual changes or synthesis tools
- If the state space is not the same, sequential (behavioral equivalence) of FSM must be considered...
• To verify the behavior of such circuits we need efficient representation for the manipulation of next-state and output functions and sets of states

• Using characteristic functions of relations and sets
Relational Representation of FSM

Representation of Relations and Sets

• If R is n-ary relation over \{0,1\} then R can be represented by (the ROBDD of) its characteristic function: \(f_R(v_1,\ldots,v_n) = 1 \iff (v_1,\ldots,v_n) \in R\)
  - Same technique can be used to represent sets of states

• Transition relation \(N\) of a sequential circuit is represented by its Boolean characteristic function over inputs and state variables:
  \[N(\mathbf{x}, y_1, \ldots, y_s, y_1', \ldots, y_s')\]

• **Example**: synchronous modulo 8 counter, \(N(\mathbf{y}, y') = N_0(\mathbf{y}, y_0') \land N_1(\mathbf{y}, y_1') \land N_2(\mathbf{y}, y_2')\)

![Diagram of a synchronous modulo 8 counter](image_url)

Next state \(y'\):
- \(y_0' = \neg y_0\)
- \(y_1' = y_0 \oplus y_1\)
- \(y_2' = (y_0 \land y_1) \oplus y_2\)

Transition relation \(N(y,y')\):
- \(N_0(y,y') = (y_0' \leftrightarrow \neg y_0)\)
- \(N_1(y,y') = (y_1' \leftrightarrow y_0 \oplus y_1)\)
- \(N_2(y,y') = (y_2' \leftrightarrow (y_0 \land y_1) \oplus y_2)\)
Quantified Boolean Formulas (QBF)

- Needed to construct complex relations and manipulate FSMs
- \( V = \{v_1, v_2, ..., v_n\} \) = set of Boolean (propositional) variables
- QBF(V) is the smallest set of formulas such that
  - every variable in V is a formula
  - if \( f \) and \( g \) are formulas, then \( \neg f, f \land g, f \lor g \) are formulas
  - if \( f \) is a formula and \( v \in V \), then \( \forall v. f \) and \( \exists v. f \) are formulas
- A truth assignment for QBF(V) is a function \( \sigma: V \to \{0,1\} \)
  
  If \( a \in \{0,1\} \), then \( \sigma[v \leftarrow a] \) represents
  
  \[
  \sigma[v \leftarrow a](w) = \begin{cases} 
  a & \text{if } v = w \\ 
  \sigma(w) & \text{if } v \neq w
  \end{cases}
  \]
- \( f \) is a formula in QBF(V) and \( \sigma \) is a truth assignment: \( \sigma \models f \) if \( f \) is true under \( \sigma \).
Relational Representation of FSM (cont’d)

Quantified Boolean Formulas (cont’d)

- QBF formulas have the same expressive power as ordinary propositional formulas; however, they may be more concise.
- QBF Semantics: relation $\models$ is defined recursively:
  
  - $o \models v \iff \sigma(v)=1$;
  - $o \models \neg f \iff \sigma \neq f$;
  - $o \models f \lor g \iff \sigma \models f$ or $\sigma \models g$;
  - $o \models f \land g \iff \sigma \models f$ and $\sigma \models g$;
  - $\sigma \models \exists v. f \iff \sigma[v\leftarrow 0] \models f$ or $\sigma[v\leftarrow 1] \models f$;
  - $\sigma \models \forall v. f \iff \sigma[v\leftarrow 0] \models f$ and $\sigma[v\leftarrow 1] \models f$.

- Every QBF formula can represent an n-ary Boolean relation consisting of those truth assignments for the variables in V that makes the formula true: Boolean characteristic function of the relation

- $\exists x. f = f|_{x\leftarrow 0} \lor f|_{x\leftarrow 1}$, $\forall x. f = f|_{x\leftarrow 0} \land f|_{x\leftarrow 1}$

In practice, special algorithms needed to handle quantifiers efficiently (e.g., on ROBDD)
Sequential Equivalence Checking

Basic Idea:
To prove the equivalence of two FSMs $M_1$ and $M_2$ (with the same input and output alphabet), a *product machine* is formed which tests the equality of outputs of the two individual machines in every state.

$M_1$ and $M_2$ are equivalent iff the product machine produces $\text{Flag} = \text{true}$ output in every state reachable from the initial state.

- Coudert *et al.* were first to recognize the advantage of representing set of states with ROBDD's: Symbolic Breadth-First Search of the transition graph of the product machine.
- Their technique was initially applied to checking machine equivalence and later extended by McMillan, et al. to symbolic model checking of temporal logic formulas (in CTL).
Relational Product of FSMs

Relational Products — implementation using ROBDD

- A typical task in verification: compute relational products with abstraction of variables:
  \[ \exists v. [f(v) \land g(v)] \]

- Algorithm \textit{RelProd} computes it in one pass over ROBDDs \( f(v) \) and \( g(v) \), instead of constructing \( f(v) \land g(v) \)

- \textit{RelProd} uses a \textit{computed table} (result cache), and is based on Shannon’s expansion

- Entries in the cache have the form \((f, g, E, h)\), where \( E \) is a set of variables that are existentially qualified out and \( f, g \) and \( h \) are (pointers to) ROBDDs

- If an entry indexed by \( f, g \) and \( E \) is in the cache, then a previous call to \textit{RelProd} \((f, g, E)\) has returned \( h \), it is not recomputed

- Algorithm works well in practice, even if it has theoretical exponential complexity
Relational Product Algorithm

\[ \text{RelProd} \left( f, g: \text{ROBDD}, E: \text{set of variables} \right) \]
if \( f=\text{false} \lor g=\text{false} \) then return false
else if \( f=\text{true} \land g=\text{true} \) then return true
else if \((f, g, E, h)\) is cached then return \( h \)
else let \( x \) and \( y \) be the top variables of \( f \) and \( g \), respectively
let \( z \) be the topmost of \( x \) and \( y \),
\( h_0:=\text{RelProd}(f|z=0, g|z=0, E) \)
\( h_1:=\text{RelProd}(f|z=1, g|z=1, E) \)
if \( z \in \Delta E \)
\( \text{then } h:=\text{Or}(h_0, h_1) \) \{ROBDD: \( h_0 \lor h_1 \}\)
else \( h:=\text{IFThenElse}(z, h_1, h_0) \)
endif
insert \((f, g, E, h)\) in cache
return \( h \)
endif
Reachability Analysis on FSMs

Computing Set of Reachable States

- Reachable state computation (state enumeration) is needed for FSM equivalence and model checking
- $S_0$ = a set of states, represented by the ROBDD $S_0(V)$

Find those states $S_1$ reachable in at most one transition from $S_0$:

ROBDD’s $S_0(y)$ and $N(y, y')$, compute an ROBDD representing $S_1$:

$S_1 = S_0 \cup \{ s' \mid \exists s [ s \in S_0 \land (s, s') \in N] \}$

$S_1(y') = S_0(y') \lor \exists y_i [ S_0(y) \land N(y, y')]$

$y_i \in y$

$S_2 = S_0 \cup \{ s' \mid \exists s \in S_1 \land (s, s') \in N \}$

$S_2(y') = S_0(y') \lor \exists y_i [ S_1(y) \land N(y, y')]$

$y_i \in y$
Reachability Analysis (cont’d)

- In general, the states reachable in at most k+1 steps are represented by:

\[ S_{k+1}(y') = S_0(y') \lor \exists y_i \left[ S_k(y) \land N(y, y') \right] \]

\[ y_i \in y \]

- As each set of states is a superset of the previous one, and the total number of states is finite, at some point, we must have \( S_{k+1} = S_k \), \( k \leq 2^s \) the number of states

- Reachability computation can be viewed as finding “least fixpoint”

- What about inputs \( x \)? Existentially quantify them out in the relational product (equivalent to closing the system with a non-deterministic source of values for \( x \))
BDD Encoding

Basic idea:
1) connect both machines to equality check of outputs
2) compute set of reachable states
   2a) representing set of states using ROBDD
   2b) computing “images” of BDDs of all next states (using transition relations)
   2c) reachability iteration (using images starting from one initial state until sequence emerges)

\[ R_0 = \text{initial BDD} \]
\[ \ldots \]
\[ R_{i+1} = R_i \lor \text{Image}(R_i) \rightarrow \text{convergence}; \]
ROBDD Encoding (cont’d)

Representing set of states using ROBDDs

- Set: {110}, {010}, {100, 101, 110, 111, 010, 011}
- Formula: $x_0 \land x_1 \land \neg x_2$, $\neg x_0 \land x_1 \land \neg x_2$, $x_0 \lor x_1$
- ROBDD diagrams for each set and formula.
Sequential Equivalence Checking Example

1) Connect both machines to equality check of outputs
2a) Representing set of states using ROBDD

Initial State:  
x0 = 0  
x1 = 1  
x2 = 0

\{0,1,0\}  

\overline{x0} \land x1 \land \overline{x2}  

\text{formula}

\text{ROBDD}
2b) Compute image of set \{0 1 0\}

set = \{0 1 0\} \quad \neg x_0 \land x_1 \land \neg x_2

\text{image:} \quad (\neg x_0 \land x_1 \land \neg x_2) \lor (x_0 \land \neg x_1 \land x_2)
\begin{align*}
i &= 0 \\
i &= 1
\end{align*}

(BDD_1 \lor BDD_0)

transition relation: \quad [y_0 = (x_0 \oplus i)] \land \\
[y_1 = (\neg i \land x_1) \lor (i \land x_2)] \land \\
[y_2 = (\neg i \land x_2) \lor (i \land x_1)]
2b) Compute images of set \{0 1 0\}

Transition Relation

\[
\begin{align*}
y_0 &= x_0 \oplus i \\
y_1 &= (\neg i \land x_1) \lor (i \land x_3) \\
y_2 &= (\neg i \land x_2) \lor (i \land x_1)
\end{align*}
\]

<table>
<thead>
<tr>
<th>(x_0)</th>
<th>(i = 0)</th>
<th>(i = 1)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(BDD1)</th>
<th>(BDD2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>((\neg x_0 \land x_1 \land \neg x_2))</td>
<td>((x_0 \land \neg x_1 \land x_2))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{{0,1,0}, {1,0,1}}</td>
<td>ROBDD</td>
</tr>
</tbody>
</table>

\[\text{disj. } BDD1 \quad BDD2\]
Example (cont’d)

2c) Reachability iteration

\[ R_0 = \neg x_0 \land x_1 \land \neg x_2 \]
\[ R_1 = (\neg x_0 \land x_1 \land \neg x_2) \lor (x_0 \land \neg x_1 \land x_2) \lor R_0 \]
\[ \rightarrow R_2 = R_1 \]

In terms of sets:

\[ R_0 = \{010\} \]
\[ R_1 = \{010, 101\} \]
\[ R_2 = \{010, 101\} \quad R_2 = R_1 \]

⇒ Converged

⇒ all states reached!
Equivalence Checking Tools

Commercial tools:
- Chrysalis: Design Verifier
- Synopsys: Formality
- Cadence: Conformal
- Verysys: Tornado
- AHL: ChekOff-E

Application:
- Used to prove equivalence of two sequential circuits that have the same state variables (or at least the same state space and a known mapping between states) by verifying that they have the same next-state and output functions
- Used in place of gate vs. RTL verification by simulation

Recommendations:
- Use modular design, relatively small modules, 10k - 20k gates
- Maintain hierarchy during synthesis (not flattening) and before layout: equivalence can be proven hierarchically much faster, especially for arithmetic circuits
Equivalence Checking Tools (cont’d)

CheckOff-E

- Commercial product by Abstract Hardware Ltd. (UK) and Siemens AG (Germany)
- Performs behavioral comparison of two Finite State Machines
- Input EDIF netlist + library or VHDL
- VHDL subset (superset of synthesizable synchronous VHDL)
  - no real time clauses (after, wait for), no conditional loop statements
- Interprets VHDL simulation semantics to build a Micro FSM
- Converts to Macro FSM by merging transition until stabilization at each time t
- Macro FSM is starting point for any verification; representation in ROBDD
- **Product discontinued!**
References


