Introduction to
Formal Verification and Logic Synthesis

Yukio Miyasaka
Contents

Formal Verification

• Equivalence Checking
  • Binary Decision Diagram
  • Boolean Satisfiability Problem

Logic Synthesis

• Two-Level Logic Synthesis
• Two-Level Logic to Multi-Level Logic
• Multi-Level Logic Optimization
Formal Verification

- Prove that a given logic circuit meets some given properties
- Often compared to random simulation, here are some tradeoffs:

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Formal Verification</th>
<th>Random Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100%</td>
<td>#test patterns / #all possible patterns</td>
</tr>
<tr>
<td>Time</td>
<td>~O(exp(circuit size))</td>
<td>O(#test patterns * circuit size)</td>
</tr>
</tbody>
</table>

Note: coverage has a different meaning for bounded model checking
Equivalence Checking

• Given two implementations, prove they are functionally equivalent
  • For simplicity, assume the implementations are combinational logic circuits
  • Optimized implementation v.s. Golden model (Spec)

• Exhaustive simulation is enough to prove this property

• However, it takes an exponential amount of time
  • N inputs -> 2^N patterns to simulate
  • O(exp(#inputs))

• Can we do it better?
Binary Decision Diagram (BDD)

• Binary tree
• Redundant nodes (equivalent cofactors) are removed
• Each node has a unique function

BDD Operations

• We can construct a BDD for the result of operation by a recursive procedure
  • e.g. BDD $z = \text{AND}(\text{BDD } x, \text{BDD } y);$
  • Runtime is bounded by $O(\#\text{BDD nodes})$ while $\#\text{BDD nodes}$ is usually smaller than $\exp(\#\text{inputs})$
Symbolic Simulation

Output BDDs to be compared with the golden model
Boolean Satisfiability Problem

- Problem definition \( \exists \vec{x}. f(\vec{x}) = 1 \).
- One of the most famous NP-complete problems
- \( f(\vec{x}) \) is usually given as CNF (Conjunctive Normal Form) a.k.a. POS
  - Variable \( x_1, x_2, ... \)
  - Literal \( x_1, x_1', x_2, x_2', ... \)
  - Clause \( x_1 + x_2' + x_3, ... \)
  - CNF \((\text{Clause } A) \cdot (\text{Clause } B) \cdot ...\)
- Many heuristics have been proposed
- Can be solved much faster than the other NP-complete problems
History of SAT Solver Improvement

No heuristics + 2-literal clause table + Restarting (Current Standard)

Clause learning

Clause learning and unlearning

SAT-based Equivalence Checking

\[ f_{\text{AND}} = (a' + b' + c)(a + c')(b + c') \]

\[ f_{\text{MITER}} \left( \frac{\exists \text{Inputs, Internal.}}{\text{Inputs, Internal, NEQ}} \right) \cdot \text{NEQ}. \]

If satisfiable, there exists a pattern where two circuits output different values.

If unsatisfiable, two circuits are equivalent.
Performance Comparison

• For i10 benchmark (257 inputs, 224 outputs, about 2000 gates)
  • Exhaustive simulation: Never ends ($2^{257} > 10^{25}$ patterns)
  • Symbolic simulation: 0.65 sec
  • SAT-based: 0.43 sec

• SAT-based method is usually faster than symbolic simulation
• Exceptions are arithmetic circuits like multipliers
Logic Synthesis
Logic Synthesis

• Generates a logic circuit from various kinds of descriptions such as truth table, Boolean expression, etc.
• Important metrics: area (#gates) and depth (#levels)
Two-Level Logic

• AND gates in the first level, OR gates in the second level
• Direct representation of SOP
• The basic strategy is the same as Karnaugh map
• However, for more than one output, we have to care logic sharing
  • $x = a + a'bc$, $y = b'c + a'bc$ is better than
  • $x = a + bc$, $y = b'c + a'c$
• ESPRESSO heuristic logic minimizer:
  • Developed by Robert K. Brayton (emeritus professor at UC Berkeley)
  • Incompatible with modern C compilers, while you can find a patched version on GitHub or other websites
**ESPRESSO: Example**

```
<table>
<thead>
<tr>
<th>test.pla</th>
<th>out.pla</th>
</tr>
</thead>
<tbody>
<tr>
<td>.i 3</td>
<td>.i 3</td>
</tr>
<tr>
<td>.o 2</td>
<td>.o 2</td>
</tr>
<tr>
<td>.p 8</td>
<td>.p 3</td>
</tr>
<tr>
<td>000 00</td>
<td>-01 01</td>
</tr>
<tr>
<td>001 01</td>
<td>011 11</td>
</tr>
<tr>
<td>010 00</td>
<td>1-- 10</td>
</tr>
<tr>
<td>011 11</td>
<td></td>
</tr>
<tr>
<td>100 10</td>
<td></td>
</tr>
<tr>
<td>101 11</td>
<td></td>
</tr>
<tr>
<td>110 10</td>
<td></td>
</tr>
<tr>
<td>111 10</td>
<td></td>
</tr>
<tr>
<td>.e</td>
<td>.e</td>
</tr>
</tbody>
</table>
```

`espresso test.pla > out.pla`
Two-Level Logic to Multi-Level Logic

• We can reduce #gates by factoring
  • \( y = b'c + a'c = (b' + a')c \)

• **Fast Extract** is a greedy algorithm for factoring
  • For each pair of products, calculate divisors
    • \( (abc, a'bc') \rightarrow \text{divisors} = \{ac + a'c', b\} \)
  • For each divisor, count how many literals it can save
  • Factor out the divisor that can save the most
  • Repeat until no more factoring is possible
  • (Some details are omitted)

Fast Extract: Example

- 5-input majority function: \( abc + abd + abe + acd + \cdots \)
  - Factoring out a single variable can save 5 literals, while sum of two variables can save 10 literals
- Factor out \( a + b \)
  - \((a + b)(cd + ce + de) + abc + abd + abe + cde\)
- Factor out \( ab \)
  - \((a + b)(cd + ce + de) + ab(c + d + e) + cde\)
- Factor out \( c + d \)
  - \((a + b)(cd + (c + d)e) + ab((c + d) + e) + cde\)
- After handling trivial cases where \( cd \) is shared, we get 12 gate implementation:
  - First level: \( n_0 = a + b, n_1 = ab, n_2 = c + d, n_3 = cd \)
  - Second level: \( n_4 = n_2e, n_5 = n_2 + e, n_6 = n_3e \)
  - Third level: \( n_7 = n_3 + n_4, n_8 = n_1n_5 \)
  - Forth level: \( n_9 = n_0n_7, n_{10} = n_8 + n_6 \)
  - Fifth level: \( n_{11} = n_9 + n_{10} \)
Multi-Level Logic Optimization

• Fast Extract is not optimal
  • Quality depends on the initial SOP
  • What if multiple divisors can save the same number of literals?
  • One of them might lead to a better result in the end

• Rewriting is one of the most popular optimization methods
  • Extract a subcircuit (4-5 inputs) iteratively
  • Replace it with an equivalent precomputed minimum circuit
  • How can we precompute the minimum circuit?
Exact Synthesis

• Encode all possible circuits of $N$ gates into a CNF
• Let a SAT solver find one that is equivalent to the specification
• If the solver cannot find one with $N-1$ gates but one with $N$ gates, the latter one is proven to be minimum
• There are many encoding methods

Exact Synthesis Encoding

Gate 1

Gate type select signals

Input select signals

All inputs

All inputs
Exact Synthesis Encoding
Exact Synthesis Encoding

Gate type select signals

Gate 1

All inputs

All inputs

Gate 2

All inputs

All inputs + Gate 1

All inputs

All inputs + Gate 1

For each output

All inputs + All gates + Constants
Exact Synthesis Encoding

\[
\exists \text{Select signals. } \forall \text{Inputs. } f \left( \text{Inputs, Select signals} \right) = \text{Spec (Inputs)}.
\]

For each output

All inputs + All gates + Constants
More About Logic Optimization

• Exact synthesis works up to 5 gates.
• Rewriting is local optimization, does not necessarily lead to the global optimal
• There are many other optimization methods
  • Merge equivalent internal nodes
  • Substitute one node with a new subcircuit
  • Compute internal don’t-cares using BDD, and perform equivalent transformation
• Phase ordering problem
  • Once you apply one optimization, some other optimizations may be no longer effective
  • The effective order varies by circuits
  • Machine learning to find a good optimization order?