Learning Boolean functions from hardware designs

Shobha Vasudevan
ECE Department
University of Illinois at Urbana-Champaign

Design verification

Specification in C, C++, English language docs

Does design satisfy asserted properties?

Implementation at logic level

Hinges on knowing what to look for
What are assertions?

Statements of desired intent in a system
Expressed in propositional or temporal logic
If an assertion fails, a bug is exposed!

1. If the microwave is started, it will eventually heat
2. If the door is not closed, the microwave cannot heat
3. If an error condition occurs, the microwave must be reset before heating
4. If the microwave is heating, then the door is closed and there is no error

What is hard about writing assertions

• Intensely manual and iterative effort
• Inter-module assertions get left out
• Temporal relations are challenging to capture
• Many assertions are synthesized on chip
  – Too many assertions: Redundant and costly
  – Too few assertions: Not enough coverage
  – Sweet spot: Takes ~40% of pre silicon design time to arrive at!
    (PC: IBM, Qualcomm)
Integrating solution spaces

Integrating Solution Spaces

Combine both to offset their disadvantages
18. Learning Boolean functions from hardware designs

GoldMine

ASSESSMENTS

Temporal Assertions
Propositional Assertions

Design
Static Analysis

Design Constraints

Likely Invariants

Simulation Traces

Data Generator

A-Miner

Counterexamples

System Invariants

Formal Verification

A-Val

Designer Feedback Loop

GoldMine highlights

Licensed by all 3 leading EDA companies
Used by universities to check Verilog designs in courses
Available for research
3 best paper awards/nominations
18 students

Unique features
- Short list of assertions
- Assertion ranking
- Statement coverage per assertion

Developed into commercial product

© Shobha Vasudevan.
18. Learning Boolean functions from hardware designs

**GoldMine**

**ASSERTIONS**
- Temporal Assertions
- Propositional Assertions

- Design
- Static Analysis
- Design Constraints
- Simulation Traces
- Likely Invariants
- System Invariants
- Counterexamples
- Designer Feedback Loop

**Composition of A-miner**

- Decision Tree
- Coverage Guided Association
- A-miner
- Decision Forest
- PRISM

© Shobha Vasudevan.
Decision tree based mining

Selects splitting variable that provides maximum information gain/minimizes entropy

Greedy approach to building minimal depth tree
Decision tree assertions

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

\[
m = 0.25 \\
H = 0.81
\]

\[
\sim a \rightarrow \sim f \\
a \& \sim b \rightarrow \sim f \\
a \& b \rightarrow f
\]

Decision forest

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

\[
n = 0.25 \\
H = 0.81
\]

\[
\sim a \rightarrow \sim f \\
\sim a \& \sim b \rightarrow \sim f \\
a \& b \rightarrow f
\]
18. Learning Boolean functions from hardware designs

Decision forest

Decision tree vs forest

Optimized to not duplicate existing orderings
Set containment in decision forest

Coverage Guided Association Mining
- Extracts all possible correlations between signals
- Uses input space coverage to guide the mining
- Coverage gain: the newly increased truth table coverage.

<table>
<thead>
<tr>
<th>Simulation trace</th>
<th>Covered truth table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c Z</td>
<td>t1 and t2 cov_gain=50%</td>
</tr>
<tr>
<td>t1</td>
<td>t3 cov_gain=25%&lt;50%</td>
</tr>
<tr>
<td>t2</td>
<td>t4 cov_gain=25%&lt;50%</td>
</tr>
<tr>
<td>t3</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>A1: a=0 =&gt; z=0</td>
</tr>
<tr>
<td></td>
<td>A2: b=0 =&gt; z=0</td>
</tr>
<tr>
<td></td>
<td>A3: c=1 =&gt; z=0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A4: a=1&amp;b=1 =&gt; z=1</td>
</tr>
<tr>
<td></td>
<td>A5: a=1&amp;b=0 =&gt; z=0</td>
</tr>
</tbody>
</table>

Solution set after Iteration 1 is A1
Solution set after Iteration 2 is A1, A2 and A4
18. Learning Boolean functions from hardware designs

Static analysis in GoldMine

- **Static analysis**
  - Word level assertion generation
  - Counterexample guidance
  - Bounded cone of influence
  - Coverage analysis of an assertion

GoldMine

- **Assertions**
  - Temporal Assertions
  - Propositional Assertions
  - Likely Invariants
  - System Invariants
  - Counterexamples
  - Designer Feedback Loop

**GoldMine**

- **Design**
  - Design Constraints
  - Simulation Traces

- **Data Generator**
- **A-Miner**
- **Formal Verification**
- **A-Val**
Decision tree based mining

Design function:
\[ z = (a \lor \neg c) \land b \]

Simulation trace:

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

Splitting variable: \( a \)

Target output:

\[ z \]

M: 0.25
H: 0.81

a = 0
M: 0
H: 0

a = 1
M: 0.50
H: 1.00

b = 0
M: 0
H: 0

b = 1
M: 1
H: 0

ASSERT1:
\( \neg \neg \neg \neg a \Rightarrow \neg \neg \neg \neg z \)

ASSERT2:
\( (a \land \neg b) \Rightarrow \neg \neg \neg \neg z \)

ASSERT3:
\( (a \land b) \Rightarrow z \)

Counterexample-based incremental decision tree

Counterexample

Temporal/Propositional Assertions

Likely Assertions

Simulation

Data Generator

Decision Tree Building

Formal Verification

Static Analysis

Target RTL Design

Simulation Traces

Incremental Decision Tree

© Shobha Vasudevan.
Incremental decision tree

Algorithm completeness and convergence
Algorithm completeness and convergence

- We formally prove the following two theorems
  - It takes finite iterations to reach the final decision tree, in which all assertions are true
  - The final decision tree corresponds to the entire functionality of the output

Design space map for an output to show functional coverage

- Initial coverage of design space is ad-hoc
Monotonically increasing coverage analysis

Every new Ctx grows the map to uncovered regions

Monotonically increasing coverage analysis

Monotonic decrease in uncovered space
Impact of the GoldMine solution

- Output directed notion of coverage
- Deterministic metric of progress in the test development process through incremental decision trees
- Strictly monotonic increase in the coverage of the test suite with every iteration
- Provably convergent for a given output
- Achieves coverage closure!

Converging coverage of tests

Coverage closure achieved within 15 iterations
Word level assertions

(a[0] = 1) ^ (b[0] = 0) => (state[0] = 1)
(a[0] = 1) ^ (b[0] = 0) => (state[1] = 0)
(a[1] = 1) ^ (b[1] = 0) => (state[0] = 1)

(a > b) => (state = active)

Word level assertion generation

1. Identify constant assignments to word level output variables
2. Discover word level targets
   - Select all word level conditional expressions in target's logic cone
   - Get conditional expressions in terms of primary inputs/register variables

RTL source code

Instrument word level predicates

Simulate augmented RTL

A-miner

Traces
18. Learning Boolean functions from hardware designs

Word level assertion generation

- Weakest liberal precondition \( wlp(st, P) \)
- Statically computed symbolic value of variable
- Complex computation due to path explosion in transitive closure
- Exact computation not required

Simulate along single concrete path for \( k \) cycles
Symbolic values of word level expressions for concrete path
Propagate until primary inputs or reg variables

Sliding window for multi-cycle simulation path

- Slide the mining window along simulation path
- Compute wlp within each window

\( n: \) simulation path length
\( m: \) mining window length

© Shobha Vasudevan, University of Illinois
Word level assertion generation

- Simulation guided weakest liberal precondition computation (up to k cycles)
- Weakest liberal precondition \( wlp(st, P) \)
- Statically computed
- Complex computation
- Exact computation
- Use concrete paths in computing symbolic value of variable
- Complexity of computation drastically reduced
- Underapproximation of \( wlp \)
- Sliding k cycle mining window is scalable

RTL design example

```verilog
module A(c, d, p, q, state)
parameter active = 2'b01;
Parameter idle = 2'b10;
input [1:0] c, d;
input [1:0] p, q;
reg [1:0] a, b;
reg [1:0] state;

always@(*)
    if(c>d)
        a = p;
    else
        a = p+1;

always@(*)
    if(c=d)
        b = q-1;
    else
        b = q+1;

endmodule
```

UD chain a
b2
b3
UD chain b
b5
b6
b1: c>d
b2: a=p
b3: a=p+1
b4: c=d
b5: c=d
b6: b=q-1
b7: b=q+1
b8:

b9: a>b
b10: state=active
b11: state=idle
b12:

Taken path: b1->b2->b4 / b5->b7->b8 / b9->b10->b12

b2 is the definition to variable a in concrete path
b7 is the definition to variable b in concrete path

Simulation guided weakest precondition

Discovered word level feature: a>b

Note: variable a and b are NOT primary inputs and should NOT used as features.

b3 is the definition to variable a in concrete path
b7 is the definition to variable b in concrete path

Simulation guided weakest precondition

Discovered word level feature: p>q+1
Initial attempts at subjective ranking

1 - Uninteresting assertion, would not use in the design
2 - True design constraint, might use to test the design
3 - Captures subtle design intent, would likely use to test the design

© Shobha Vasudevan.
Subjective Designer Rankings

Designer gives a higher percentage of coverage
guided mining assertions a rank 3

© Shobha Vasudevan.
Number of Assertions

Assertion evaluation

Importance

Readability

Complexity

Coverage

Expectedness
Assertion evaluation

Importance

Readability

Assertion Importance

Google PageRank

© Shobha Vasudevan, University of Illinois
 Assertions Importance

\[ PR(p) = \sum_{p_i \in B(p)} \frac{PR(p_i)}{|F(p_i)|} \]

- \( PR(p) \): rank of page \( p \)
- \( B(p) \): set of pages that link to \( p \)
- \( F(p) \): set of pages that \( p \) links to

Assertion readability

\[ C(a) = c_k^{(|a| - 1)} \]

- \( C(a) \): readability of assertion \( a \)
- \( 0 < c_k < 1 \): “as readable as” constant
- \(|a|\): number of propositions in \( a \)’s antecedent

- \( X \land Y \land Z \) is “nearly as readable as” \( X \land Y \)
- Readability decreases sharply with increase in antecedent size
- Experiments show maximum agreement with human judgment at 0.75
### Assertion Complexity

- **Understandability**
- **Temporal and spatial distance considered**
- **Relative to consequent variables**

```verilog
counter = 3:00;  
c_at = 2:00;  

gnt  
c_at = 0:00;  

16 always @(*)  
17 begin  
18    gnt1 = req1 & ~req2;  
19    gnt2 = req2;  
20 end  
21 else  
22 begin  
23    gnt1 = req1;  
24    gnt2 = req2 & ~req1;  
25 end  
26 end
```

### PCI Master State Machine

- **Executes requests to target devices**
  - Acquire control of the bus
  - Broadcast the target address and command
  - Transfer data between the host and target
GoldMine assertion for PCI

\[ a1: \text{cur\_state}[3] == 1 \] \#1 \ ( \text{rdy\_in} == 0 \) \Rightarrow \ ( \text{pci\_frame\_en\_out} == 0 ) \]

If the state machine's current state is turn around one cycle later the host is not ready to send data, then disable pci_frame_out

Highest Ranked Assertion

\[ \text{cur\_state}[3] == 1 \]

The state machine's next state is idle

\[ a1: \text{cur\_state}[3] == 1 \] \#1 \ ( \text{rdy\_in} == 0 \) \Rightarrow \ ( \text{pci\_frame\_en\_out} == 0 ) \]

If the state machine's current state is turn around and one cycle later the host is not ready to send data, then disable pci_frame_out
Highest Ranked Assertions

rdy_in == 0
• frame_en_slow == 0 since the state machine’s current state is idle
• frame_en_keep == 0 since the state machine’s current state is idle

```
13 wire frame_en_slow = (sm_idle & u_have_pci_bus & req_in & rdy_in) ||
14 sm_address || (sm_data_phases & ~pci_frame_out_in);
15 wire frame_en_keep = sm_data_phases & pci_frame_out_in & ~mabort1 & ~mabort2;
16 assign pci_frame_en_out = frame_en_slow ||
17 frame_en_keep & pci_stop_in & pci_trdy_in;
18
```

a1: (cur_state[3] == 1) ##1 (rdy_in == 0) |-> (pci_frame_en_out == 0)
If the state machine’s current state is turn around and one cycle later the host is not ready to send data, then disable pci_frame_out

Highest Ranked Assertions

pci_frame_en_out == 0
Disable pci_frame_out

```
13 wire frame_en_slow = (sm_idle & u_have_pci_bus & req_in & rdy_in) ||
14 sm_address || (sm_data_phases & ~pci_frame_out_in);
15 wire frame_en_keep = sm_data_phases & pci_frame_out_in & ~mabort1 & ~mabort2;
16 assign pci_frame_en_out = frame_en_slow ||
17 frame_en_keep & pci_stop_in & pci_trdy_in;
18
```

a1: (cur_state[3] == 1) ##1 (rdy_in == 0) |-> (pci_frame_en_out == 0)
If the state machine’s current state is turn around and one cycle later the host is not ready to send data, then disable pci_frame_out
module pci_master32_sm(
  input clk_in, reset_in, pci_gnt_in, pci_frame_in,pci_frame_out_in, pci_irdy_in, pci_trdy_in,
  pci_stop_in, req_in, rdy_in,
  output pci_frame_out, pci_frame_en_out);

reg sm_idle;
reg sm_address;
reg sm_data_phases;
reg sm_turn_arround;

wire u_dont_have_pci_bus = pci_gnt_in || ~pci_frame_in || ~pci_irdy_in;
wire u_have_pci_bus = ~pci_gnt_in && pci_frame_in && pci_irdy_in;

wire frame_en_slow = (sm_idle && u_have_pci_bus && req_in && rdy_in) || sm_address || (sm_data_phases && ~pci_frame_out_in);
wire frame_en_keep = sm_data_phases && pci_frame_out_in && ~mabort1 && ~mabort2;
assign pci_frame_en_out = frame_en_slow || frame_en_keep && pci_stop_in && pci_trdy_in;

reg [3:0] cur_state;
reg [3:0] next_state;

parameter S_IDLE = 4'h1;
parameter S_ADDRESS = 4'h2;
parameter S_TRANSFER = 4'h4;
parameter S_TA_END = 4'h8;

wire ch_state_slow = sm_address || sm_turn_arround || sm_data_phases && (pci_frame_out_in && mabort1 || mabort2);
wire ch_state_med = ch_state_slow || cur_state[0] == 1;

Highest Ranked Assertions

a1: ( cur_state[3] == 1 ) ##1 ( rdy_in == 0 ) |->( pci_frame_en_out == 0 )
If the state machine’s current state is turn around and one cycle later the host is not ready to send data, then disable pci_frame_out
Highest Ranked Assertions

In the next cycle, the state machine’s current state will be idle

4 output pci_frame_out, pci_frame_en_out);
5 reg sm_idle;
9
10 wire u_dont_have_pci_bus = pci_gnt_in || ~pci_frame_in || ~pci_irdy_in;
11 wire u_have_pci_bus = ~pci_gnt_in && pci_frame_in && pci_irdy_in;
12
13 wire frame_en_slow = (sm_idle && u_have_pci_bus && req_in && rdy_in) ||
14 sm_address || (sm_data_phases && ~pci_frame_out_in);
15 wire frame_en_keep = sm_data_phases && pci_frame_out_in && ~mabort1 && ~mabort2;
16 assign pci_frame_en_out = frame_en_slow ||
17 frame_en_keep && pci_trdy_in && pci_stop_in;
18
22 parameter S_IDLE = 4’h1;
23 parameter S_ADDRESS = 4’h2;
24 parameter S_TRANSFER = 4’h4;
25 parameter S_TA_END = 4’h8;
26
28 wire ch_state_slow = sm_address || sm_turn_arround ||
29 sm_data_phases && (pci_frame_out_in && mabort1 || mabort2);
30 wire ch_state_med = ch_state_slow || sm_idle && u_have_pci_bus && req_in && rdy_in;
31 wire change_state = ch_state_med ||
32 sm_data_phases && (~pci_trdy_in && pci_stop_in);
33
a1: (cur_state[3] == 1) ##1 (rdy_in == 0) |-> (pci_frame_en_out == 0)
If the state machine’s current state is turn around and one cycle later the host is not ready to send data, then disable pci_frame_out
Highest Ranked Assertions

\[
\text{pci\_frame\_en\_out} = 0
\]

Disable \text{pci\_frame\_out}

\[
\text{a1: (cur\_state[3] == 1 )##1 (rdy\_in == 0 )|-> (pci\_frame\_en\_out == 0 )}
\]

If the state machine's current state is turn around and one cycle later the host is not ready to send data, then disable \text{pci\_frame\_out}