8. Sequential Equivalence Checking

J. A. Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin
EE 382M-11, Verification of Digital Systems
Fall 2010
February 25, 2010

Implementation Verification

Verify the equivalence of two designs (possibly at different levels, such as RTL vs. Gate levels)

Note: This will not detect bugs at the higher (RT) level
Implementation Verification of Sequential Circuits

- Given a netlist, does it correctly implement the design?
- State machine comparison techniques are applicable to a large class of designs through several stages of the design process when behavior of the design can be represented by a state machine
  - The design may be specified as a scheduled control/data flow graph at the highest level
- State machine comparisons can also be used for design verification
- Drawbacks
  - Implementation has to retain precise input-output correspondence with specification
  - State space explosion for large machines
- If state elements at both levels have the identical encoding, verification reduces to checking equivalence of combinational circuits (approach used in the commercial verification tools)

Approaches to Sequential Equivalence Checking

Reduce the FSMs to combinational circuits

- Unroll FSM over $n$ time frames (flatten the design)
- Check equivalence of resulting combinational circuits
- Problem: circuit may be too large to handle

Note: This approach can deal with two machines which have different state encodings
Approaches to Sequential Equivalence Checking, Cont’d

Check isomorphism of state transition graphs
- Two machines are equivalent if their state transition graphs are isomorphic
- Minimize the two state machines to be checked for equivalence
- Check if the two graphs are isomorphic
- Problem: State transition graphs of realistic machines can be too large to represent (300 state variables ⇒ more nodes in the state transition graph than the number of protons in the universe)

Symbolic traversal of the product FSM
- Create product FSM (M) of the two machines to be checked ($M_1$ and $M_2$)
- Traverse the states of the product machine and check its output for every transition
- If all the outputs are equal ($O(M)=1$), $M_1$ and $M_2$ are equivalent
- Otherwise, an error trace can be found
Approaches to Sequential Equivalence Checking, Cont’d

Language Containment
- Model: finite-state automata that accept sequences
  - For specifications use an \( \omega \)-automaton (L-automaton \( T \))
  - For implementations use a non-deterministic FSM (L-process \( A \))
- Verification is testing for language containment \( L(A) \subset L(T) \)
- A transition between states is enabled by a set of input values
  - Predicates representing these sets form a Boolean algebra \( L \)
- The transition structure of the automaton/machine is represented by an adjacency matrix over \( L \), where the \((i, j)\) entry specifies the enabling predicate for a transition from state \( i \) to state \( j \)
- The transition L-matrix which may be nondeterministic

Reducing the Complexity in Language Containment

- Decomposition
  - Given an \( \omega \)-regular language \( L \) a finite number of deterministic L-automata \( \Gamma_1, \Gamma_2, \ldots, \Gamma_n \) can be found such that \( L = \bigcap_{i=1}^{n} L(\Gamma_i) \)
  - Now verification is to test for \( L(A) \subset L(\Gamma_i) \) for all \( i = 1, \ldots, n \)
- Reduction: an implementation description \( A \) is transformed to \( A' \) such that
  \[
  L(A') \subset L(T') \implies L(A) \subset L(T)
  \]
  where \( T' \) is the corresponding reduction of the original specification \( T \)
State Enumeration Techniques

- Very important to sequential synthesis, verification, testing, extraction of sequential don’t cares
- Explicit Enumeration
  - Complexity depends on the number of states and input values
  - Process one state at a time
  - For each state, next states are enumerated one by one
  - Example is Murφ (Murphi) from Stanford
- Implicit Enumeration
  - Set of states are manipulated in a breadth-first fashion
  - Exponential speed up over explicit enumeration methods
  - May take a long time for sequentially deep circuits

Representing Sequential Circuits Using ROBDDs

- ROBDDs can represent synchronous sequential circuits symbolically
- Consider system modeled as a Mealy FSM which is a 6-tuple: 
  \((\Sigma, O, S, S_0, \delta, \lambda)\)
- Cannot explicitly build the State Transition Graph for any practical system
  - Represent set of states as predicates expressed as ROBDDs
  - Such an ROBDD depends only on the \(n\) Primary State Variables (PSVs) and we avoid dealing directly with the \(2^n\) possible states of the machine
  - No need to specify the encodings of the various states
  - State Transition Graph treated as a predicate transformer

Department of Electrical and Computer Engineering, The University of Texas at Austin
Jacob A. Abraham, February 25, 2010
Representing Circuits Using ROBDDs (Cont’d)

- Create a Boolean variable for each node in $V$ the set of state holding nodes and $\Sigma$
  - A state is described by a valuation assigning 0 or 1 to each variable
  - denote sets of states with $S$ and the ROBDD representing the set $S$ by $S(V)$
- Use ROBDDs to represent the transitions that the system can make
  - Create a second set of variables $V'$.
  - A valuation for the variables in these two sets can be considered to be designating an ordered pair of states in the circuit
  - Such sets of pairs of states are called transition relations
  - If $N$ is a transition relation, we denote the ROBDD representing it as $N(V, V', \Sigma)$
- Overall transition relation: $N(V, V', \Sigma) = \bigwedge_{i=0}^{n-1} N_i(V, V', \Sigma)$

A Simple Example

- Consider a synchronous modulo–8 counter, and let $V = v_0, v_1, v_2$ and $V' = v'_0, v'_1, v'_2$
- The transitions of the modulo–8 counter are given by the following equations:
  
  $v'_0 = \neg v_0$
  $v'_1 = v_0 \oplus v_1$
  $v'_2 = (v_0 \land v_1) \oplus v_2$

- The above equations can be used to define the relations
  
  $N_0(V, V') = (v'_0 \iff \neg v_0)$
  $N_1(V, V') = (v'_1 \iff v_0 \oplus v_1)$
  $N_2(V, V') = (v'_2 \iff (v_0 \land v_1) \oplus v_2)$

- Transition relation is given by
  
  $N(V, V') = N_0(V, V') \land N_1(V, V') \land N_2(V, V')$
Reachability Analysis

- Reachable state computations are at the heart of any formal verification approach for sequential circuits
- States reachable from $S_0$
  \[ S_1 = S_0 \cup \{ s' | \exists s [s \in S_0 \land (s, s') \in N] \} \]
  - Using ROBDDs:
    \[ S_1(V') = S_0(V') \lor \exists v \in V \lor \Sigma [S_0(V) \land N(V, V', \Sigma)] \]
- Computations can be viewed as finding a least fixed point
- Predicate transformer $F$:
  \[ F(S) = S_0 \cup \{ s' | \exists s [s \in S \land (s, s') \in N] \} \]
  - Using ROBDDs we have:
    \[ F(S)(V') = S_0(V') \lor \exists v \in V \lor \Sigma [S(V) \land N(V, V', \Sigma)] \]
  - We have $F(S_i)(V') = S_{i+1}(V')$
  - The sequence of state sets $0, F(0), F^2(0), \text{etc.}$, converges to the least fixed point of $F$ under the set containment ordering, and the fixed point is exactly the set of reachable states
Example

Consider a vector $f : \mathbb{B}^2 \rightarrow \mathbb{B}^2$ of Boolean functions as

\[ f := [f_1, f_2], \text{ where } f_1(x_1, x_2) = x_1 \lor x_2 \text{ and } f_2(x_1, x_2) = x_1 \oplus x_2 \]

\begin{array}{|c|c|c|}
\hline
x_1 & f_1(x_1, x_2) & f_s(x_1, x_2) \\
\hline
F & F & F \\
F & T & T \\
T & F & T \\
T & T & F \\
\hline
\end{array}

Let $C := \{[F,F], [T,F]\}$

Image Operators

- Given a set $Z \subseteq S$, define the following operators:

\[ \text{Image}(N, Z) = \{v | \exists u [u \in Z \land N(u, v)] \} \]

\[ \text{PreImage}(N, Z) = \{u | \exists v [v \in Z \land N(u, v)] \} \]

\[ \text{BackImage}(N, Z) = \{u | \forall v [N(u, v) \Rightarrow v \in Z] \} \]
Quantification Operations

Existential Quantification  \( \exists v[f] = f|_{v=0} + f|_{v=1} \)

Universal Quantification  \( \forall v[f] = f|_{v=0} \cdot f|_{v=1} \)

\[
f = \exists x_2. (x_1 x_2 + \overline{x_1} x_3 x_4)
\]

Image Operators Using ROBDDs

- Image:
  \[
  T(V') = \exists v \in V \cup \Sigma [S(Z) \land N(V, V', \Sigma)]
  \]
  \[
  I_{N,Z}(V) = \forall v' \in V' [\exists v \in V [(v \iff v') \land T(V)]]
  \]

- Preimage:
  \[
  T(V') = \forall v \in V [\exists v' \in V' [(v \iff v') \land S(Z)]]
  \]
  \[
  T_1(V, \Sigma) = \forall v' \in V [T(V') \otimes N_i(V, V', \Sigma)]
  \]
  \[
  PI_{N,Z}(V) = \exists \sigma \in \Sigma [T_1(V, \Sigma)]
  \]

- Backimage:
  \[
  T(V') = \forall v \in V [\exists v' \in V' [(v \iff v') \land S(Z)]]
  \]
  \[
  T_1(V, \Sigma) = \forall v' \in V [T(V') \otimes N_i(V, V', \Sigma)]
  \]
  \[
  PI_{N,Z}(V) = \forall \sigma \in \Sigma [T_1(V, \Sigma)]
  \]
Equivalence Checking as a State Reachability Problem

Counterexample Generation
Example

A modulo-three counter (one input $x$, two state variables, $q_1$, $q_2$)

Next state functions $(\delta_1, \delta_2)$, with
\[
\delta_1 := \neg q_1 q_2 x \lor q_1 \neg q_2 \neg x
\]
and
\[
\delta_2 := q_2 \neg x \lor \neg q_1 \neg q_2 x
\]

Transition relation
\[
R(q, q') = \exists x.((q'_1 \leftrightarrow \delta_1(q_1, q_2, x)) \land (q'_2 \leftrightarrow \delta_2(q_1, q_2, x)))
\]
\[
= \exists x.((q'_1 \leftrightarrow (\neg q_1 q_2 x \lor q_1 \neg q_2 \neg x)) \land (q'_2 \leftrightarrow (q_2 \neg x \lor \neg q_1 \neg q_2 x)))
\]

Characteristic function for the initial state, $C(q) = \neg q_1 \neg q_2$

Then, $\text{img}(\delta, C) = \exists q. R(q, q') \land C(q) = q'_1$

The state $(1,1)$ is not reachable from the starting state

Can also use backward traversal from this state (e.g., a bad state)

Sequential Equivalence Checking of System Level and RTL Descriptions

- Unprecedented levels of integration on a chip
- SoC verification is a formidable challenge
  - Functional verification consumes 50-75\% of design cycle
  - Classical simulation techniques or hardware-software co-verification techniques cannot handle the size and complexity of SoCs
- New verification approach for tackling functional and structural complexity of SoCs

Formal Equivalence Checking

Equivalence checking can be applied at a given level or across levels

Challenges in Sequential Equivalence Checking

- Challenge: Verifying two designs with different state encodings
- Many sequential equivalence techniques have been proposed
  - Sequential SAT solvers
  - Use of ATPG techniques
  - Latch mapping methods
- All techniques operate at gate level and reason in Boolean domain

Look at ways of verifying large designs
Sequential equivalence checking technique between System-level models (SLM) and RTL
Very little research in SLM vs RTL equivalence checking
EDA vendors like Calypto provide a partial solution using bounded model checking for SLM vs RTL equivalence checking
Intuition of Dealing With the Problem at a Higher Level

- Boolean level techniques involve explicit state space reasoning
  - State-transition matrices are intractable, as they deal with Boolean values of variables
- High level reasoning encapsulates more information per fundamental entity than BDDs
- Revise notion of “state”: more complex artifact
  - RTL state: Aggregate of (consecutive) gate states
  - Transitions between RTL states: Concatenation of gate transitions
  - Transitions between System Level states: Concatenation of RTL transitions

Static Analysis to Efficiently Decompose Design Spaces

- Leverage information available at System and Register Transfer Levels
  - Use expressiveness of design entities in the source code of these levels
- Introduce a theoretically sound decomposition technique
  - Splitting equivalence checking problem space
  - Sequential compare points that exactly model sequential behavior
- Technique used in conjunction with lower level engines
  - Creates more tractable problems
  - Assists in overcoming capacity issues
Sequential Compare Points

- Compare points are used in combinational equivalence checking
  - Used both for flip-flop outputs as well as internal combinational nodes (to reduce sizes of blocks being verified)
- Sequential compare points are two-tuple entities
  - Identification with respect to relative position in time
  - Identification with respect to space (data or variables)
- Co-ordinates on space-time axis of both designs being compared
- Exactly model the sequential behavior of designs
- Variables of interest (observables) obtained from user/block diagram
  - Typically include primary outputs
  - Can also include relevant intermediate variables
- Symbolic expressions obtained for observables assigned in a given cycle
- Symbolic expressions compared at sequential compare points

Computing the Symbolic Expression

- To compute a symbolic expression for $v$ at $t$
  - Substitute L.H.S definition for $v$ in assignment
  - Include control information necessary for substitution to be valid
  - Include cycle $t$ where value is computed (identifier)
  - Repeat procedure transitively, until input or previously “observed” variable is reached
- Symbolic expression at a given compare point remains same for all future compare points
  - All future references need not expand the observables at a previous compare point
  - Savings when using a lower level Boolean engine
Algorithm

M: System level model
V: RTL model
O: list of observables

Construct the behavioral state transition graph for both M and V

Obtain all variables that are assigned to in one transition X

For all transitions X and for all time t

Compute symbolic value for variable v at time t in both models

For all variables v

Concatenate symbolic value for variable v over all time

Check equivalence of symbolic value for variable v using a SAT solver

If satisfiable

Obtain Proof

Error Trace

Example System C and Verilog Code

```c
SC_module (bcd_to_bin) { 
    sc_in clk clk_i;
    sc_in<bool> rst_i;
    sc_in<sc_uint<5>> dat_i;
    sc_out<sc_uint<5>> dat_o;
    sc_uint<4> msb;
    sc_uint<4> ls_byte;
    void convert_code() {
        msb = dat_i[4].read();
        ls_byte = dat_i[3:0].read();
        dat_bin_o.write((msb*10)+byte);
    }
}
```

```verilog
module bcd_to_bin (clk_i, dat_i, dat_o);
    input clk_i;
    input [4:0] dat_i;

    output [3:0] dat_o;
    reg [3:0] dat_o;
    always @(posedge clk_i) begin
        tens_digit <= dat_i[3:0] * 4'd10;
        no_tens_digit <= dat_i[3:0];
        tens_select <= dat_i[4];
    end
    always @(posedge clk_i) begin
        if (tens_select) dat_o <= tens_digit;
        else dat_o <= no_tens_digit;
    end
endmodule
```
Example, Cont’d

Comparison point
C1 = (t=2, d= dat_o)

Symbolic expression in System C for dat_o

\[ dat_o(t+1) = (dat_i[4](t+1) \times 10) + dat_i[3:0](t+1) \]

Symbolic expression in Verilog for dat_o

\[ dat_o(t+2) = dat_i[4](t+1) ? (dat_bcd_i[3:0](t) + 4’d10) : dat_i[3:0](t) \]

Check for the equivalence of the two expressions for dat_o

Can use any low-level engine to achieve this

Using a SAT solver in this work

Generating Symbolic Expressions – Verilog Code and State Transition Graph

```verilog
always @(clk)
begin
    if (reset) {
        a = 0;
        b = 0;
    } else {
        if (sel) {
            a = 1;
            b = 0;
        } else {
            a = 0;
            b = 1;
        }
    }
    result[0] = a;
    result[1] = b;
end
```
result is the observable signal, assigned in the second clock cycle

First iteration – at time $t$:
result[0] = a and result[1] = b,
$E_t = f(V[result[1]/b, result[0]/a], (reset = 0), t)$
$E_t = t \land (reset = 0) \land result[1]/b \land result[0]/a$

Second iteration – at time $t - 1$:
$E_{t-1} = ((t - 1) \land (reset = 0) \land (sel = 0) \land (a/0) \land (b/1)) \lor ((t - 1) \land (reset = 0) \land (sel = 1) \land (a/1) \land (b/0))$

Third iteration – at time $t - 2$:
$E_{t-2} = (t - 2) \land (reset = 1) \land (a = 0) \land (b = 0)$

Symbolic Expression $E$ is obtained by a logical AND operation of $E_t, E_{t-1}$ and $E_{t-2}$

Example: Viterbi Decoder

- Part of software radio, Digital Radio Mondiale (DRM) in SystemC
- DRM SoC partitioned to implement Viterbi decoder as a hardware accelerator
- SystemC specification
  - Basic model implementing Viterbi algorithm
  - No optimizations
- Viterbi Verilog RTL implementations
  - First implementation: Optimized for speed
  - Second implementation: Optimized for area
Viterbi Decoder: SystemC Specification

Viterbi Decoder Implementation 1
Proof Setup

- List of observables
  - 8 FIFO entries, each 32-bits wide: FF[7:0][31:0]
  - 64 Trellis Matrix entries, each 32-bits wide: TM[63:0][31:0]
  - 2 entries in the MatDec, each 32-bits wide: MD[1:0][31:0]
  - Decoded output, 32-bits wide: Out[31:0]
- Signal mapping between two designs provided
- Both designs started in reset state
- System C design output computed every cycle
- Verilog implementation 1 output computed every 10th cycle
- Verilog implementation 2 output computed every 16th cycle

Decomposing Equivalence Checking between SystemC and Implementation

Department of Electrical and Computer Engineering, The University of Texas at Austin
Jacob A. Abraham, February 25, 2010
Outline of Steps to Check Equivalence

Start with both the designs at the reset state initially, and step the two designs in tandem.

Output in the SystemC model is computed every cycle, output in the Verilog model is computed in the $10^{th}$ cycle after the reset state.

First set of observables FF[7:0][31:0] is available after 8 cycles.

The first compare point, $C_1 = (t = 8, d = FF[7:0][31:0])$

The two symbolic expressions for $FF_s[i][31:0]$ and $FF_v[i][31:0]$ are checked using a SAT solver for the 8 entries in the FF Buffer.

After the $8^{th}$ cycle, the Verilog design assigns to the Trellis Matrix in the $10^{th}$ cycle.

The second compare point, $C_2 = (t = 10, d = TM[63:0][31:0], MD[1:0][31:0], Out[31:0])$

This is continued till all the observables are checked.
Checking Symbolic Expressions with \textit{zchaff}

- Translate check() into a Boolean satisfiability problem
- XNOR operation used to combine target symbolic expressions
- Without decomposition Trellis Matrix generates 1.9 million clauses!
  - Chokes \textit{zchaff}
- Using sequential compare points
  - 32 independent CNF formulas
  - Each formula contains 59136 clauses
  - \textit{zchaff} is able to run on each formula

Viterbi Decoder Implementation 2
Decomposition of Equivalence Checking Between System C and Implementation Optimized for area

Checking Design Optimized for Area

\[ C_1 = (t = 8, d = FF[7 : 0][31 : 0]) \]
\[ C_2 = (t = 10, d = TM[15 : 0][31 : 0], MD[1 : 0][7 : 0]) \]
\[ C_3 = (t = 12, d = TM[31 : 16][31 : 0], MD[1 : 0][15 : 8]) \]
\[ C_4 = (t = 14, d = TM[32 : 47][31 : 0], MD[1 : 0][16 : 23]) \]
\[ C_5 = (t = 16, d = TM[47 : 63][31 : 0], MD[1 : 0][23 : 31], Out[31 : 0]) \]
### Results: Number of Variables and Clauses for SAT Solver

<table>
<thead>
<tr>
<th>Block/Function</th>
<th>Number of clauses in the CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>448</td>
</tr>
<tr>
<td>LESSTHAN</td>
<td>32</td>
</tr>
<tr>
<td>Trellis Condition in the Butterfly</td>
<td>14336</td>
</tr>
<tr>
<td>Trellis computation in each stage of butterfly</td>
<td>28672</td>
</tr>
<tr>
<td>Trellis per butterfly</td>
<td>57344</td>
</tr>
<tr>
<td>MatDec each stage of butterfly</td>
<td>896</td>
</tr>
<tr>
<td>MatDec per butterfly</td>
<td>1792</td>
</tr>
</tbody>
</table>

### Number of Variables and Clauses for SAT Solver, Cont’d

<table>
<thead>
<tr>
<th>Block/Function</th>
<th>Number of variables</th>
<th>Number of Symbolic variables generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Butterfly</td>
<td>128</td>
<td>66</td>
</tr>
<tr>
<td>Trellis (monolithic)</td>
<td>2304</td>
<td>2112</td>
</tr>
<tr>
<td>Trellis (decomposed)</td>
<td>128</td>
<td>66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Number of clauses in the CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic Trellis</td>
<td>1892352</td>
</tr>
<tr>
<td>RTL decomposition (Design 1)</td>
<td>59136</td>
</tr>
<tr>
<td>RTL decomposition (Design 2)</td>
<td>59136</td>
</tr>
</tbody>
</table>