Sequential Equivalence Checking

EE 382M-11
The University of Texas at Austin
Department of Electrical and Computer Engineering

Dr. Xiushan “Shaun” Feng
Formal Verification Group Leader,
Samsung Austin Research Center (SARC), Austin, TX
Samsung Advanced Computing Lab (ACL), San Jose, CA
Samsung G2 Design Center, Gyeonggi-do, Korea

Outline

- Sequential Equivalence Checking (SEC) basics
  - Combinational EC vs. Sequential EC
  - Theory behind SEC
    - Reachability analysis on product machine
    - Bounded vs. unbounded, etc.
    - Partition on transition relation, CFG, etc.
- How Industry deal with SEC issues
  - Leverage existing CEC and model checking tools
  - Sequential equivalence checking tools
    - E.g., SLEC, JasperGold SEC, VCF-SEQ, Hector, ESP-CV (RTL vs. Spice)
**Equivalent Checking - Miter**

For all possible aligned inputs, whether outputs are always equivalent
- G and R can have different state elements
- G and R can have different latencies.

**CEC (State Points Mapped)**

- All combinational circuits are aligned -- CEC
  - EC over outputs of combo circuits, i.e., next-state functions of state elements (induction prove of EC)
  - Good scalability, matured, extensively used.
- Requires: complete state mappings
**Types of SEC**

- Input/output alignment
  - Cycle-accurate equivalence
    - Equivalent at every cycle
  - Transaction-level equivalence
    - Compare points can have different latencies

- Initial state
  - Safe replacement (Singhal, Pixley, Aziz, Brayton)
    - No assumption about the initial state
  - Initial state needed
    - From init state, whether non-eq states can be forward reached
    - Or from non-eq states, where init state can be backward reached

---

**Common SEC in Verification**

- Electronic System Level (ESL)
  - ESL vs ESL (e.g., Matlab vs. timed SystemC)
  - ESL vs RTL (e.g., serial C vs. RTL, EC of HLS)

- RTL vs. RTL – commonly used
  - Pipeline updates
  - Register retiming
  - Resource rescheduling
  - State recoding
  - Sequential clock gating verification
  - Xprop verification

- RTL vs. Spice (switch-level)
Example: Clock Gating SEC

<table>
<thead>
<tr>
<th>CG_en==0</th>
<th>CG_en==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other inputs</td>
<td>Other inputs</td>
</tr>
<tr>
<td>Instance d0 of DUT</td>
<td>Instance d1 of DUT</td>
</tr>
<tr>
<td>- cg clocks always running</td>
<td>- cg clocks can be disabled based on internal state</td>
</tr>
</tbody>
</table>

- For all possible input combinations, there is no output mismatch up to certain cycles
- If possible, full prove for all cycles
- d0.out_foo == d1.out_foo
  or d0.out_valid && d1.out_valid |\rightarrow| d0.out_foo == d1.out_foo

Example: XPROP with SEC

<table>
<thead>
<tr>
<th>All Inputs</th>
<th>All Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance d0 of DUT</td>
<td>Instance d1 of DUT</td>
</tr>
<tr>
<td>- X assignments</td>
<td>- X assignments</td>
</tr>
<tr>
<td>- Uninitialized registers</td>
<td>- Uninitialized registers</td>
</tr>
<tr>
<td>- Array range overflow</td>
<td>- Array range overflow</td>
</tr>
<tr>
<td>- Multiple drivers, etc..</td>
<td>- Multiple drivers, etc..</td>
</tr>
</tbody>
</table>

- Xs inside RTL are don't-care spaces (Synthesis/formal chooses 0 or 1)
- For all possible input combinations, there is no X propagated to outputs
- The reason that two instances are not equivalence is due to different assignments of Xs
- d0.out_foo == d1.out_foo
  or d0.out_valid && d1.out_valid |\rightarrow| d0.out_foo == d1.out_foo
**CEC vs SEC**

Taken from reference 1

![Diagram showing differences between CEC and SEC](image)

<table>
<thead>
<tr>
<th>CEC</th>
<th>SEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-encoding of state</td>
<td></td>
</tr>
<tr>
<td>Serial vs parallel interfaces</td>
<td></td>
</tr>
<tr>
<td>Scheduling</td>
<td></td>
</tr>
<tr>
<td>Pipelining</td>
<td></td>
</tr>
<tr>
<td>FFs match</td>
<td></td>
</tr>
</tbody>
</table>

**SEC Approaches**

- **Flattening**
  - flatten a sequential circuit into a combinational circuit
  - Reduce SEC into CEC --- too big to handle

- **Graph isomorphism**
  - Two FSMs can be translated into the same one
  - Re-writing rules, canonical forms.

- **Reachability analysis on product machine**
  - Turn SEC into model checking on equality assertions
  - State space explosion
    - Bounded model checking (DAC’03, Kroening, Clarke, and Yorav)
    - Abstraction techniques (slicing, exploiting similarity, etc.)
    - Redundancy removing
Flatten

Sequential circuit $C$

Unroll the sequential circuit by $t$ time frames

Combinational circuit

Isomorphic State Graph

Rewriting $G_{\text{min}}$ and $R$ to check for equivalence.

SEC -- Model Checking

- Product machine: \( M = G \times R \)
- Assertions: \( G_{\text{out}} == R_{\text{out}} \)
- Reachability
  - For all reachable states, whether \( G_{\text{out}} == R_{\text{out}} \)
    - For all initial states?
    - Or for a given initial state?
  - Termination criteria:
    - fixpoint reached. e.g., least fixpoint if init state is given, no new state visited
    - Upper bound hit in BMC (bounded model checking)

Monolithic Transition Relation – BDD\(^*\) Example

```c
/* Builds a single BDD that's the transition relation for the entire circuit. */
for (i=0; i<state_var_count; i++) {
    /* Build the relation for each next state wire. */
    wire_rel = Cudd_bddXnor(gbm,
                            next_array[i]->bdd_var,
                            next_array[i]->bdd);
    // predicate of \( x' = f(x) \)
    Cudd_Ref(wire_rel);
    temp = Cudd_bddAnd(gbm,TR,wire_rel);
    Cudd_Ref(temp);
    Cudd_RecursiveDeref(gbm,TR);
    Cudd_RecursiveDeref(gbm,wire_rel);
    TR = temp;
}
```

- Monolithic TR is built
  - one big BDD for TR
  - BDDs for flops are ANDed into one.

\(^*\)CUDD is used (http://vlsi.colorado.edu/~fabio/CUDD/)
Post Image Computation – BDD Example

/* Computes AND and quantifies out present state and input variables */
   temp = Cudd_bddAndAbstract(gbm,
      S,
      TR,
      ps_input_cube);
   Cudd_Ref(temp);

   /* Now, change the image BDD back to present state variables. */
   post_S = Cudd_bddSwapVariables(gbm,
      temp,
      ps_vars,
      ns_vars,
      state_var_count);
   Cudd_Ref(post_S);
   Cudd_RecursiveDeref(gbm,temp);

- Image computation for all reachable states
  - reachable states are represented by one big BDD.

Least Fixpoint Computation – BDD Example

/*Least Fixpoint (LFP) Computation. Loop terminates when LFP is reached*/
S = Cudd_ReadLogicZero(gbm);  Cudd_Ref(S);
new_R = build_initial_state_bdd(); // a user function to set initial state.
Cudd_Ref(new_R);
do {
   temp= Cudd_bddOr(gbm,S, post_S);
   Cudd_Ref(temp);
   Cudd_RecursiveDeref(gbm,S);
   Cudd_RecursiveDeref(gbm,post_S);
   S = temp;
   post_S = image_monolithic_tr(TR, S, ps_in_cube, ps_vars, ns_vars);
   Cudd_Ref(post_S);
} while (S!=post_S);

- LFP works on reachable states
  - Monolithic transition relation
  - One BDD for reachable states
State Space Explosion

- SEC works on 2 times design space (G + R)
  - Exponentially increased state space.
- Model checking cannot handle large designs or complicated arithmetic circuits.
  - BDD: size
  - SAT: number of clauses, runtime
- Abstraction techniques are needed.

SEC Abstractions

- Simplify model G and R
  - Rewriting – make G and R more alike
  - Redundancy removing – drop unneeded logic
  - Retiming – move logic across state points.
- Divide and conquer – decomposition
  - Partition transition relations, state space, flop stage, etc.
- Use correspondences between G and R to simplify the product machine
  - Structure similarities between G and R.
TBV: Transformation-Based Verification

- Design N
- Redundancy Removal Engine
- Result N

- Design N'
- Retiming Engine
- Result N'

- Design N''
- Target Enlargement Engine
- Result N''

- Design N'''
- Result N'''

Taken from reference 3 on IBM SixthSense

TBV: Redundancy Removal Results

- Number of Registers
- IFU
- SMM
- S6669

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Original Design</th>
<th>After Merging via Induction</th>
<th>After Merging via TBV</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6669</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Taken from reference 3 on IBM SixthSense
Partition monolithic transition relation by assignment on each variable to reduce the size of SAT clauses. (Reference 7)
Annotated Control Flow Graph

- ACFG is a Partitioned Model Checking method
  - Partition the software states and hardware states based on the structure of ACFG
- Reduce the state space
  - Use the flow graph and antecedents of ACFG to guide and tailor the state space exploration
- Idea:
  - Given the flow, and antecedents → what set of states can be on each edge
  - After the computation → consequents are checked against result
ACFG Partition Algorithm 1

1: function ModelCheck($ACFG$, $post_c$

(* Incorporate mapped antecedents to the circuit and
initialize the simulation relation *)
2: for all edges $e$ of $ACFG$ graph do
3: $\text{ant}(e) := \text{ant}(e) \land \text{ant}(e)[I_g/I_c][O_g/O_c]$;
4: if $e$ is from the entry vertex then
5: $\text{sim}(e) := post(e)(\text{ant}(e))$;
6: add $e$ into taskQueue;
7: else
8: $\text{sim}(e) := \emptyset$;
9: end if
10: end for

ACFG Partition Algorithm 2

(* Compute simulation relation and check consequents *)
11: while taskQueue $\neq \emptyset$ do
12: remove an edge $e$ from taskQueue;
13: if $\text{sim}(e) \nRightarrow \text{cons}(e)$ then
14: return(a counter-example trace);
15: end if
16: for each successor edge $e'$ of $e$ do
17: $\text{sim}(e') := \text{sim}(e') \lor$

$\text{post}(e)(\text{post}_c(\text{sim}(e)) \land \text{ant}(e'))$;
18: if there is a change in $\text{sim}(e')$ then
19: put $e'$ into taskQueue;
20: end if
21: end for
22: end while
23: return(succeed);
ACFG: Results

- We take radix-2 SRT divider as an example (2N-bit dividend, N-bit divisor)

<table>
<thead>
<tr>
<th>N</th>
<th>Time(s)</th>
<th>BDD size</th>
<th>Time(s)</th>
<th>BDD size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12.27</td>
<td>5390</td>
<td>6.99</td>
<td>7462</td>
<td>1.76x</td>
</tr>
<tr>
<td>8</td>
<td>32.31</td>
<td>10307</td>
<td>18.41</td>
<td>14635</td>
<td>1.76x</td>
</tr>
<tr>
<td>9</td>
<td>21.89</td>
<td>8982</td>
<td>14.97</td>
<td>9770</td>
<td>1.46x</td>
</tr>
<tr>
<td>10</td>
<td>50.62</td>
<td>6619</td>
<td>26.26</td>
<td>15566</td>
<td>1.93x</td>
</tr>
<tr>
<td>11</td>
<td>392.66</td>
<td>10915</td>
<td>379.52</td>
<td>59125</td>
<td>1.03x</td>
</tr>
<tr>
<td>12</td>
<td>727.43</td>
<td>19722</td>
<td>484.12</td>
<td>29792</td>
<td>1.50x</td>
</tr>
<tr>
<td>13</td>
<td>1854.62</td>
<td>63555</td>
<td>877.29</td>
<td>38756</td>
<td>2.11x</td>
</tr>
<tr>
<td>14</td>
<td>950.25</td>
<td>22262</td>
<td>636.86</td>
<td>44919</td>
<td>1.50x</td>
</tr>
<tr>
<td>15</td>
<td>452.57</td>
<td>20036</td>
<td>193.19</td>
<td>56982</td>
<td>2.34x</td>
</tr>
</tbody>
</table>

Similarity

- Signal correspondence TCADICS00, C. A. J. van Eijk
  - Internal state point mapping
  - Possible equivalent internal signals (wire, flop)
  - Cutpoint insertion
- Text similarity IWLS’03, Matsumoto, Saito, and Fujita
  - If minor diffs between two RTL, focus on diffs
- Control flow in ESL vs. control logic in RTL
  - Merging points in ESL to find RTL correspondence (software cutpoint, EMSOFT05 DAC06, Feng and Hu)
Leverage Similarities

- If multiple stages of flops can be proven equivalent, we can do assume-guarantee.

Cutpoints and Blackboxes
CutPoint Abstraction Theory

- for $\forall x f_1(x) = g_1(x)$
  - if $\forall z, y f_2(z, y) \equiv g_2(z, y)$, then $f_2(f_1(x), y) \equiv g_2(g_1(x), y)$ $\Rightarrow F \equiv G$
  - if $\exists z, y f_2(z, y) \neq g_2(z, y)$ cannot say $F \equiv G$

- Cutpoint theory is equivalent to
  - Uninterpreted function in SMT and TRS (Term Rewriting System)
  - Blackboxing in CEC/SEC

False non-equivalence:
The tool reports not equivalent when designs are equivalent
- CEC can have false non-eq due to cutpoints
- Constraints are needed to remove infeasible space

Assume-Guarantee Reasoning of Cutpoints

- Double-side cutpoint
  - Prove $f_1(x) \equiv g_1(x)$
  - Cut $F.z$ and $G.z$
  - assume $F.z \equiv G.z$

- Single-side cutpoint
  - Prove $f_1(x) \equiv g_1(x)$
  - Cut $G.z$
  - Assume $G.z \equiv f_1(x)$
**Software Cutpoint**

[DAC’06, EMSOFT’05, Feng and Hu]

- Preliminary static analysis
  - Dependence analysis
  - Dataflow analysis: live variables analysis
  - Identify path merging points

- Formal equivalence checking
  - Linearly unroll loops
    - Merge paths based on preliminary analysis
  - Reduce logic blow-up
  - Cutpoint insertion
### Path Enumeration vs. Linear BDD

<table>
<thead>
<tr>
<th>Example</th>
<th>Path Enumeration</th>
<th>Linear Building BDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX20-8</td>
<td>241.24 28</td>
<td>0.28 61</td>
</tr>
<tr>
<td>EX20-16</td>
<td>time out</td>
<td>89.01 1746</td>
</tr>
<tr>
<td>EX20-32</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX20-64</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX97-8</td>
<td>4229.44 183</td>
<td>1.46 92</td>
</tr>
<tr>
<td>EX97-16</td>
<td>time out</td>
<td>1187.72 1800</td>
</tr>
<tr>
<td>EX97-32</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX97-64</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX251-12</td>
<td>time out</td>
<td>309.18 1843</td>
</tr>
<tr>
<td>EX251-16</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX251-32</td>
<td>time out</td>
<td>mem out</td>
</tr>
<tr>
<td>EX251-64</td>
<td>time out</td>
<td>mem out</td>
</tr>
</tbody>
</table>

### Linear BDD vs. Early Cutpoints

<table>
<thead>
<tr>
<th>Example</th>
<th>Linear Building BDD</th>
<th>Early Cutpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX20-8</td>
<td>0.28 61</td>
<td>0.11 58</td>
</tr>
<tr>
<td>EX20-16</td>
<td>89.01 1746</td>
<td>0.24 60</td>
</tr>
<tr>
<td>EX20-32</td>
<td>mem out</td>
<td>0.53 64</td>
</tr>
<tr>
<td>EX20-64</td>
<td>mem out</td>
<td>1.35 72</td>
</tr>
<tr>
<td>EX97-8</td>
<td>1.46 92</td>
<td>0.51 64</td>
</tr>
<tr>
<td>EX97-16</td>
<td>1187.72 1800</td>
<td>1.10 73</td>
</tr>
<tr>
<td>EX97-32</td>
<td>mem out</td>
<td>2.35 95</td>
</tr>
<tr>
<td>EX97-64</td>
<td>mem out</td>
<td>5.41 136</td>
</tr>
<tr>
<td>EX251-12</td>
<td>309.18 1843</td>
<td>0.64 66</td>
</tr>
<tr>
<td>EX251-16</td>
<td>mem out</td>
<td>1.09 71</td>
</tr>
<tr>
<td>EX251-32</td>
<td>mem out</td>
<td>7.45 170</td>
</tr>
<tr>
<td>EX251-64</td>
<td>mem out</td>
<td>16.81 327</td>
</tr>
</tbody>
</table>
**SEC at Industry**

- **Leverage existing formal verification tools**
  - CEC resolves a subset by flattening/remodeling
  - Model checking tool + assertions
    - Small block level design
    - Cycle accurate equivalence

- **Apply SEC tools**
  - Calypto SLEC
  - JasperGold SEC
  - IBM SixthSense
  - Synopsys ESP-CV (RTL vs Spice), VC Formal SEQ, Hector
  - In-house tools

---

**SLEC Frontend Architecture**

Taken from reference 1 and 2

- SystemC
- Sys.Verilog
- Verilog
- language X
- CPT API
- CPT
- CPT to CDB xforms
- CDB
- CDB API
- SLEC Verification Engine
- Future Products
- ESL Synthesis Engine

- Loop Unrolling
- Dependence Analysis
- Flop/Mux inferencing
- Constant Propagation
- Dead code elimination
- Smart memory modeling

- Language Neutrality
  - Support multiple languages scalably
  - Language independent transforms
SLEC Verification Engine Architecture

- Name based mappings
- Structural Decomposition
- Orchestration
- Proof Decomposition
- Simulation Engine
- Machine Acceleration
- Sequential Analysis
- Convergence Analysis
- Inductive Analysis
- Fixed point Analysis
- BLS
- Solver
- WLS
- SAT
- BDD
- Simulation
- IPBDP
- WSAT

Taken from reference 1 and 2

SLEC: Throughput and Latency

- Interface and compare points alignment
  - How golden and revised synchronized?
  - When to compare?

Fig taken from reference 2
SLEC – Setup and Operation

- Specification vs. Implementation

![Diagram showing specification vs. implementation](image)

ESP-CV

- SEC on RTL vs. Spice (switch-level)
- Symbolic simulation (1, 0, X, Z, S) on product machine
- Bounded by simulation cycles or memory size
  - Terminates when simulation cycles complete.
  - Mem over limit, randomly picks values for symbols.
- Exploits symmetry inside macro (SRAM, ROM) models – efficient models for G and R to reduce size of symbolic expression
What We’ve learned

- SEC basics
  - Flattening
  - Interface alignment
  - Reachability analysis on product state machine
- Some research ideas
  - Redundancy removal based on TBV
  - Partitioning
  - Software cutpoint insertion
- Industry tools – SLEC, ESP-CV

References

1. Developing a commercially viable sequential equivalence checker. Anmol Mathur, Calypto Design Systems,
2. SLEC user manual, Calypto
4. ESP user manual, Synopsys
8. Sequential equivalence checking based on structural similarities C. A. J. van Eijk July, 2000 TCADICS