Sequential Equivalence Checking - IV

Virendra Singh
Associate Professor
Computer Architecture and Dependable Systems Lab.
Dept. of Electrical Engineering
Indian Institute of Technology Bombay
viren@ee.iitb.ac.in

EE 709: Testing & Verification of VLSI Circuits
Lecture – 20 (Feb 14, 2012)
Symbolic FSM Traversal

- Implicit representation
- Graphs and their traversal are converted to Boolean functions and Boolean operations
- BDD can be used for symbolic computation
Symbolic FSM Representation

- $M(Q, \Sigma, \delta, q_0, F)$
- Characteristic Function

\[Q(r) = \begin{cases} 1, & r \in S \\ 0, & r \notin S \end{cases}\]

- Transition Function

\[T(p, n, a) = \begin{cases} 1, & n = \delta(p, a) \\ 0, & \text{otherwise} \end{cases}\]

\[N(s) = \begin{cases} 1, & \exists(p, a)(T(p, s, a) \cdot P(p) = 1) \\ 0, & \text{otherwise} \end{cases}\]
Symbolic FSM Traversal

Relational representation of transition function

\[ P(p_1, p_2) = \overline{p_1} \overline{p_2} + p_1 p_2 \]

\[ s_2 = p_1 + a_2 \]
Symbolic FSM Traversal

Transition relation

\[ T(p_1, p_2, s_1, s_2, a_1, a_2) = (s_1 \oplus (s_2 + \overline{a_1}a_2(p_1p_2)))(s_2 \oplus (p_1 + a_2)) \]

Next state

Present state: 00, Input: 10

\[ T(0,0,s_1,s_2,0,1) = S_1 \cdot S_2 \]

11

Set of all next state for all possible inputs

\[ N(s_1, s_2) = \exists(a_1, a_2, p_1, p_2)T(p_1, p_2, s_1, s_2, a_1, a_2)P(p_1, p_2) \]
Symbolic Model Checking

- set of all next states if the present state is either 00 or 11

characteristic function

\[ P(p_1, p_2) = \overline{p}_1 \overline{p}_2 + p_1 p_2 \]

Next state

\[ N(s_1, s_2) = \exists (a_1, a_2, p_1, p_2) T(p_1, p_2, s_1, s_2, a_1, a_2) P(p_1, p_2) \]

\[ \exists (a_1, a_2, p_1, p_2) T(p_1, p_2, s_1, s_2, a_1, a_2) P(p_1, p_2) \]
\[ = (T(p_1, p_2, s_1, s_2, 0, 0) + T(p_1, p_2, s_1, s_2, 0, 1) + T(p_1, p_2, s_1, s_2, 1, 0) + T(p_1, p_2, s_1, s_2, 1, 1))P(p_1, p_2) \]
Symbolic FSM Traversal

\[ \text{N}(S_1, S_2) = S_1 + S_2 \]

Next States: 01, 10, 11
Forward Reachability

Forward Reachable States by Symbolic Computation

Input: transition relation $T(p, s, a)$ and initial state $I(s)$

Output: a characteristic function $R(s)$ of all reachable states

ReachableState($T, I$):

1. Set $S = I$

2. Compute $N(s) = \exists (p,a)(T(p,s,a) \cdot S(p))$

3. $R = S + N$

4. If $R \neq S$, set $S = R$ and repeat steps 2 and 3; otherwise, return $R$. 
Forward Reachability

- BDD is used
Symbolic Model Checking

Forward Faulty State Reachability Analysis

Input: transition relation \( T(p, s, a) \), initial state \( I(s) \), and a fault state \( F(s) \)

Output: a resolution on whether any faulty state is reachable

\[
\text{FaultyStateReachability}(T, I, F):
\]

1. Set \( S = I \).
2. If \( (S \cdot F) \neq 0 \), return YES.
3. Compute \( N(s) = \exists (p, a)(T(p, s, a) \cdot S(p)) \).
4. \( R = S + N \).
5. If \( R \neq S \), set \( S = R \) and repeat steps 2 through 5; otherwise, return NO.
Solving Seq. EC as Comb. EC

CIRCUIT 1
PI → Comb. Logic → PO
PS₁ → MATCH → NS₁

CIRCUIT 2
PI → MATCH → PO
PS₂ → Comb. Logic → NS₂

Comb. Logic 1
PI → Comb. Logic 1 → PO
PS → Comb. Logic 1 → NS

Comb. Logic 2
PI → Comb. Logic 2 → PO
PS → Comb. Logic 2 → NS

Combinational Equiv. Checking
Combinational EC with BDD and SAT

Represent logic functions for this entire circuit with BDD or SAT
Methods for Latch Mapping

• Incomplete Methods
  – Regular expression-based using latch names
  – Using simulation (Cho & Pixley ‘97):
    • Group latches with identical simulation signatures
  – Group based on structural considerations e.g. cone of influence
  – Incomplete run of complete method below (Anastasakis et al DAC ‘02)

• Complete Methods
  – Functional fixed-point iteration based on Van Eijk’s algorithm (van Eijk ’95)
Van Eijk’s Method for Latch Mapping

Initial Latch Mapping Approximation

Apply Latch Mapping Assumptions

Comb. Logic 1

Comb. Logic 2

PI

PS

NS

Verify Latch Mapping Assumptions

Fixed-point ?

No

Iterate

Yes

Done !!

Assumptions

Apply Latch Mapping

Feb 14, 2012

EE-709@IITB
Van Eijk’s Method for Latch Mapping

init.:

\[ v_1 = 1, \]
\[ v_2 = 1, \]
\[ v_6 = 1, \]
\[ x_t = 1 \]

<table>
<thead>
<tr>
<th>( v_i )</th>
<th>init.</th>
<th>( f_i )</th>
<th>( \delta_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>1</td>
<td>( v_1 )</td>
<td>( \frac{v_1 v_2}{x_t} )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>1</td>
<td>( v_2 )</td>
<td>( \frac{v_1 v_2}{x_t} )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>1</td>
<td>( v_1 v_2 )</td>
<td>( \frac{v_1 v_2 x_t}{x_t + v_6} )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>1</td>
<td>( v_1 v_2 x_t )</td>
<td>( \frac{v_1 v_2 x_t x_{t+1}}{x_t + v_6 x_{t+1}} )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>0</td>
<td>( x_t + v_6 )</td>
<td>( x_t + v_6 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>1</td>
<td>( v_6 )</td>
<td>( x_t + v_6 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>1</td>
<td>( v_6 x_t )</td>
<td>( x_t + v_6 x_{t+1} )</td>
</tr>
</tbody>
</table>
Move to System-Level Design

System-level design

Architecture exploration

Manual effort

Barrier to adoption of System-level design Methodology!

RTL

Current design iterations

Gate-level design

Errors!
System-Level Verification

Manual effort

RTL

Gate-level design

Property & Model Checking

Equivalence Checking
System-level Synthesis

- System-level design
- Manual effort
- RTL
- Gate-level design

Automatic Synthesis

Feb 14, 2012
“The task of proof construction is in general quite tedious and a good deal of ingenuity may be required to organize the proof in a manageable fashion.

We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic.

The global state graph of the concurrent systems can be viewed as a finite Kripke structure and an efficient algorithm can be given to determine whether a structure is a model of a particular formula (i.e. to determine if the program meets its specification).”
The Model Checking Problem

The Model Checking Problem (CE81):

- Let $M$ be a Kripke structure (i.e., state-transition graph).
- Let $f$ be a formula of temporal logic (i.e., the specification).
- Find all states $s$ of $M$ such that $M, s \vDash f$
Advantages of Model Checking

- No proofs!!
- Fast (compared to other rigorous methods such)
- Diagnostic counterexamples
- No problem with partial specifications
- Logics can easily express man concurrency properties

Safety Property:
bad state unreachable

Counterexample
Main Disadvantages

• Proving a program helps you understand it. Bogus!

• Temporal logic specifications are ugly. Depends on who is writing them.

• Writing specifications is hard. True, but perhaps partially a matter of education.

• State explosion is a major problem. Absolutely true, but we are making progress!
Temporal Logic

Temporal logics describe the ordering of events in time without introducing time explicitly.

They were developed by philosophers for investigating how time is used in natural language arguments.

Most have an operator like $Gf$ that is true in the present if $f$ is always true in the future.

To assert that two events $e_1$ and $e_2$ never occur at the same time, one writes $G(: e_1 \subset : e_2)$.

The meaning of a temporal logic formula is determined with respect to a labeled state-transition graph or Kripke structure.

• Hughes and Creswell 77
Temporal Logic and Program Verification

Burstall 74, Kroeger 77, and Pnueli 77, all proposed using temporal logic for reasoning about computer programs.

Pnueli 77 was the first to use temporal logic for reasoning about concurrency.

He proved program properties from a set of axioms that described the behavior of the individual statements.

The method was extended to sequential circuits by Bochmann 82 and Owicki and Malachi 81.

Since proofs were constructed by hand, the technique was often difficult to use in practice.
Lamport was the first to investigate the expressive power of various temporal logics for verification.

His 1980 POPL paper discussed two logics: a simple linear-time logic and a simple branching-time logic.

Branching-time logic could not express certain natural fairness properties that can easily expressed in the linear-time logic.

Linear-time logic could not express the possibility of an event occurring at sometime in the future along some computation path.

Technical difficulties with method that Lamport used for his results (somewhat like comparing "apples and oranges").

Emerson and Halpern fixed these problems in an 83 POPL paper.

The temporal logic model checking algorithms of Clarke and Emerson 1980's allowed this type of reasoning to be automated.

Checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models.

The algorithm of Clarke and Emerson for CTL was polynomial in $|M|$ and in $|f|$.

We also showed how fairness could be handled without changing the complexity of the algorithm.
Clarke, Emerson, and Sistla (83 / 86) devised an improved algorithm that was linear in the product of the $|M|$ and $|f|$.

The algorithm was implemented in the EMC model checker and used to check a number of network protocols and sequential circuits.

Could check state transition graphs with between $10^4$ and $10^5$ states at a rate of about 100 states per second for typical formulas.

In spite of these limitations, EMC was used successfully to find previously unknown errors in several published circuit designs.

- EMC tool
- Fairness Constraints
- Emptiness of Non-deterministic Buchi Automata
Hardware Verification


First use of Model Checking for Hardware Verification

(found bug in the Sietz FIFO Queue from Mead and Conway, *Introduction to VLSI Systems*).

• Mishra and Clarke 83; Browne, Clarke, and Dill 86; Dill and Clarke 86
Big Events in Model Checking since 1990

- **Timed** and **Hybrid** Automata
- Model Checking for **Security Protocols**
- **Bounded** Model Checking
- Localization Reduction and **CEGAR**
- **Compositional** Model Checking and Learning
- **Infinite** State Systems (e.g., pushdown systems)
Challenges for the Future

- Software Model Checking, Model Checking and Static Analysis
- Model Checking and Theorem Proving (PVS, STEP, SyMP)
- Exploiting the Power of SAT, Satisfiability Modulo Theories (SMT)
- Probabilistic Model Checking
- Efficient Model Checking for Timed and Hybrid Automata
- Interpreting Counterexamples
- Coverage (incomplete Model Checking, have I checked enough properties?)
- Scaling up even more!!
Thank you