Sequential Verification

Scott J Weber
219 Lecture Notes
Verification of FSM Equivalence

- **Goal:** Verify that two sequential circuit implementations produce the same outputs given the same inputs

Does $\text{out1} = \text{out2}$ for all possible input sequences?
Why are state enumeration techniques important?

- State traversal techniques are essential to:
  - synthesis of sequential circuits
  - verification of sequential circuits
- such as
  - sequential testing [Cho, Hachtel and Somenzi ITC '91]
  - extraction of sequential don't care
  - implementation verification:
    - checking sequential equivalence [Coudert and Madre ICCAD '90]
      [Touati, Lin, Brayton and Sangiovanni ICCAD '90]
  - design verification:
    - checking language containment of $\omega$-regular languages [Touati, Kurshan and Brayton 1991]
    - CTL model checking [Burch, Clarke, McMillan and Dill DAC '90]
Explicit FSM Traversal

- Explicit enumeration
  - process one state at a time
  - for each state, next states are enumerated one by one
  - complexity depends on the number of states and the number of input values.
  - methods that enumerates the states space explicitly cannot handle FSM's with a large number of states
Implicit FSM Traversal

- Implicit enumeration
  - manipulates set of states directly in a BFS manner
  - exponential speed-up over explicit enumeration methods (sometimes)
  - exploits the fact that the number of states does not usually reflect system complexity
  - implicit state enumeration may take a long time for sequentially deep circuits e.g. those with counters

one set of state at a time

one step
**Transition and Output Relations**

- Let \( x \) be the present state, \( i \) the inputs, \( y \) the next state, and \( o \) the outputs.
- Transition function vector: \( f(x,i) = [f_1(x,i), ..., f_m(x,i)] \)
- Output relation:
  \( O(x,i,o) = 1 \Leftrightarrow \) output is \( o \) at state \( x \) and input \( i \)
- Transition relation:
  \( T(x,i,y) = 1 \Leftrightarrow y = \) next state from state \( x \) on input \( i \).

Note: transition relation can represent non-deterministic FSM but transition functions cannot.
Transition and Output Relations

\[ T(x, i, y) = \bigoplus_{i=1}^{m} (y_i = f_i(x, i)) \]
Range, Domain, Image and Inverse Image

- \( R(f) = \text{range} \) of \( f \) and \( D(f) = \text{domain} \) \( f(x) \) is a unique point \( y \in R \), but \( f^{-1}(y) \) is in general a set.

- The \textit{image} of a function \( f(x) \) with respect to a domain constraint \( C(x) \) is denoted
  \[
  \text{Img}(f,C) = \{ y \mid \exists x \in C : y = f(x) \}.
  \]
  In terms of relation \( T(x,i,y) \) and BDD operators:
  \[
  \text{Img}(T,C)(y) = \exists x(C(x) \cdot T(x,i,y)).
  \]

- The \textit{inverse image} of a function \( f(x) \) with respect to a domain constraint \( C(y) \) is denoted
  \[
  \text{Pre}(f,c) = \{ x \mid \exists y \in C : y = f(x) \}.
  \]
  In terms of relation \( T(x,i,y) \) and BDD operators:
  \[
  \text{Pre}(T,C)(x) = \exists y(C(y) \cdot T(x,i,y)).
  \]

- \( \text{Img}(T,C) \) and \( \text{Pre}(T,C) \) are sets represented by BDD's.
Generalized Cofactor

- Recall:
  - If $c$ is a cube, the Shannon cofactor is the cover of the generalized cofactor $\text{cof}(f,c)$ that is independent of variables of the cube $c$.
  - There are many other functions which cover $\text{cof}(f,c)$.
  - Want: heuristic to select a representative function of $\text{cof}(f,c)$ that has a small BDD representation.
  - The constrain operator [Coudert ICCAD '90] is a representation of generalized cofactor which, in most cases, has a small BDD representation.
  - When $c$ is a cube, constrain operator gives the same result as the usual Shannon cofactor.
Constrain Operator

- function `constrain(f, c)`
  - assert\(c \neq 0\)
  - if \(c = 1\) or \is\_constant(f)\) return \(f\)
  - else if \(c_{x_i} = 0\), return \constrain(f_{x_i}, c_{x_i})\)
  - else if \(c_{x_i} = 0\), return \constrain(f_{x_i}, c_{x_i})\)
  - else return \(x_i \constrain(f_{x_i}, c_{x_i}) + x'_i \constrain(f_{x'_i}, c_{x'_i})\)
Restrict Operator

- function restrict(f, c)
  
  assert(c ≠ 0)
  if (c = 1 or is_constant(f)) return f
  else if (cᵢ = 0), return restrict(fᵢ, cᵢ)
  else if (cᵢ = 0), return restrict(fᵢ, cᵢᵢ)
  else if (fᵢ = fᵢᵢ), return restrict(fᵢ, cᵢ + cᵢᵢ)
  else return xᵢ restrict(fᵢ, cᵢᵢ) + xᵢᵢ restrict(fᵢ, cᵢᵢ)

- Restrict has the property that if f is independent of xᵢ, then restrict (f, c) is also. However, it does not have the other nice properties of constrain. Experiments show that generally, restrict is better than constrain in terms of producing a smaller BDD.
Properties of Constrain Operator

- Following properties of the constrain operator make it very useful in implicit image computations. We denote \(\text{constrain}(f,c)\) by \(f_c\).
  1. \((g \circ f)_c = g \circ f_c\)
     In particular, \((f + g)_c = f_c + g_c\) and \((f')_c = (f_c)'\).
  2. \(\exists x \ f(x,y) \cdot c(x) = \exists x \ f_c(x,y)\)
  3. \(c\) is contained in \(f\) iff \(f_c\) is a tautology.
  4. If \(c\) is the characteristic function of a set \(A \subseteq B^n\), then \(f_c(B^n) = f(A)\). i.e. \(R(f_c) = Img(f,A)\)
  5. Given \(F(f,d,r)\), \(f_d\) usually has a smaller BDD representation than \(f\).
**Constrain Operator: Another Perspective**

- Let $c : B^n \rightarrow B$ be a non-null Boolean function, and $x_1 < x_2 < \ldots < x_n$ an ordering of its input variables. The mapping $\pi_c$ is defined as follows:

$$
\pi_c(x) = \begin{cases} 
1 \text{ if } c(x) = 1 \\
0 \text{ if } c(x) = 0, \text{ where} \\
\gamma = \arg\min_{y \in \text{ON-set of } c} \sum_{i=1}^n \mid x_i - y_i \mid 2^{-n}
\end{cases}
$$

- $\pi_c$ is a projection that maps a minterm $x \in B^n$ to a minterm $y$ in the ON-set of $c$, which has the closest distance to $x$, according to the distance metric $d$.

- The particular metric used guarantees the uniqueness of $y$ in the definition above.

- Since $\pi_c$ maps all points in $B^n$ into the ON-set of $c$, then $f_c(x) = f(\pi_c(x))$. If $f = (f_1, \ldots, f_m)$, then $f_c = ((f_1)_c, \ldots, (f_m)_c)$.  

13
**Fixed Point Iteration via Predicate Transformers**

- Let \((p, q)\) be two predicates on a finite set \(S\). Let \(F\) be a predicate transformer, i.e. \(F(p) = r\) where \(r\) is a new predicate on \(S\). Suppose \(F\) is monotone:
  \[ p \subseteq q \rightarrow F(p) \subseteq F(q) \]

We associate a sequence \(\{p_i\}\) as follows:

\[
\begin{align*}
p_0 &= 0 \\
p_{i+1} &= F(p_i) \\
p_\infty &= p_i \text{ if } p_{i+1} = p_i \\
\end{align*}
\]

\(p_\infty\) is called the least fixed point of the predicate transformer \(F\).

- Note that \(p_i \subseteq p_{i+1}\), since \(0 \subseteq F(p_0) \rightarrow F(0) \subseteq F(F(0))\). Since \(S\) is finite, \((p_i)_{i \geq 0}\) converges after a finite number of iterations.
**Fixed Point Iteration via Predicate Transformers**

- If $p_0 = S$, the corresponding iteration gives the *greatest fixed point* of $F$.
- All sets can be represented by BDD’s, and the predicate transformer $F(p_i)$ can be computed using BDD operations.
FSM Equivalence on the Product Machine
**FSM Equivalence of the Product Machine**

- Form the product machine:
  
  \[ x = (x_1, x_2) \]
  
  \[ y = (y_1, y_2) \]
  
  \[ i = i_1 = i_2 \]
  
  \[ Init(x) = (Init_1(x_1) \times Init_2(x_2)) \]
  
  \[ T(x, i, y) = T_1(x_1, i, y_1) \cdot T_2(x_2, i, y_2) \]
  
  \[ f(x, i) = (f_1(x_1, i) \equiv f_2(x_2, i)) \]
  
  where \( \equiv \) denotes XNOR operation.
Implicit State Enumeration

- \( T(x,i,y) = T_1(x_1,i,y_1) \cdot T_2(x_2,i,y_2) \)
- \( f(x,i) = (f_1(x_1,i) \equiv f_2(x_2,i)) \)

- The following fixed point algorithm (using a monotone predicate transformer) computes the set of reachable state pairs \( R(x) \) of the product machine:
  \[ R_0(x) = \text{Init}(x) \]
  \[ R_{k+1}(x) = R_k(x) + [y \rightarrow x] \exists x, i (R_k(x) \cdot T(x,i,y)) \]
  where \([y \rightarrow x]\) represents substitution of \( x \) for \( y \).

- The iteration terminates when \( R_{k+1}(x) = R_k(x) \) and the least fixed point is reached.
Implicit State Enumeration

- The FSM's are equivalent iff for all inputs, their outputs are identical on all reachable state pairs. Equivalence is verified by the following predicate:
  \[ \forall x \in R(x), \forall i, f(x,i) \equiv 1 \]

Easy computation with BDDs. Simply compute
\[ R(x) + f(x,i) \]
and check if this is identically 1.
Finding Reachable States in an FSM

\( R_0 \) is the set of initial states
Finding Reachable States in an FSM

$R_1$ is the set of states reachable from the initial states in less than or equal to one steps.
Finding Reachable States in an FSM

$R_2$ is the set of states reachable from the initial states in less than or equal to two steps.
Finding Reachable States in an FSM

The fixed-point iteration terminates after finding $R_4 = R_5$ and the resultant set is the set of reachable states.
Partial Product Method

- Do not need to compute the transition relation $T(x,i,y)$ explicitly as a single monolithic BDD.
- Can keep $T(x,i,y)$ in product form (called “partitioned translation relation”):

$$
\exists x.T(x,i,y) \cdot R_y(x) = \exists x. \prod_{i \in \text{in}} (y_i = (f_i)) R_y(x)
$$
Using constrain

- Can form the constrain with respect to $R_k(x)$ of each ($y_j = f_j$) before forming the final AND. This usually reduces the size of the final AND.

$$\exists x, \prod (y_j = (f_j)) R_k(x) = \exists x, \prod (y_j = (f_j))_{k(x)}$$

- The final AND operation can be decomposed into a balanced binary tree of Boolean ANDs and intermediate variables can be quantified out as soon as no other product depends on those variables.
Using restrict

- Can pre-cluster groups of $T_j(x_j,i_j,y_j)$ (defined as $y_j = f_j(x_j,i_j)$) together
  
  $C_j = \exists i_{k_j}(T_{j_1},...,T_{j_k})$

  where the $i_{k_j}$ that we can quantify out here are the ones that do not appear in any other $T_{R_{i-1}}$

- Using constrain is a bad idea since $(C_j(x_j,i_j,y_j))_{R_k(x)}$ may result in a BDD with more variables. It is better to use restrict).

- Linearly order (heuristically) the $C_j$ and multiply from the right by $R_k(x)$ one at a time, quantifying variables $x_q$, $i_q$ as soon as they appear in no other terms.)

  $R_{i+1} = R_i(x) + [y \rightarrow x] \exists x_p, i_pC_{l_j}(...)\exists x_q, i_qC_{l_q}P_i(x)\ldots$

  where $P_i(x) = \text{restrict}(R_i, R_{k-1})$. 
Traversals by Recursive Image Computation

- The method uses CONStrAIN which transforms an image computation into a range computation.
- Let $g : B^n \rightarrow B^m$, $g = [g_1, g_2, ..., g_m]$, and $A \subseteq B^n$, and $c$ be the characteristic function of $A$.
- First step: use the CONStrAIN operator to form $f = g_c = [(g_1)_c, (g_2)_c, ..., (g_m)_c]$.
- Second step: the range of $f$ is determined. Let $R(f)$ denote the range of the function $f$.
- Using Shannon's decomposition, the range computation can be written recursively as:
  \[ R(f)(y) = y_1 R([f_2, ..., f_m]_{f_1}) + y'_1 R([f_2, ..., f_m]_{f'_1}) \]
Travers*al by Recursive Image Computation

- The efficiency of the procedure can be improved by caching the results of intermediate computations. Also, if at any step, the functions \([f_1, \ldots, f_m]\) can be grouped into sets with disjoint support, then the range computation can proceed independently on each set.