Today

- Sequential Verification
  - FSM equivalence
  - Issues
    - Extracting STG
    - Valid state reduction
    - Incomplete Specification

Motivation

- Write at two levels
  - Java prototype and VHDL implementation
  - VHDL specification and gate-level implementation
- Write at high level and synthesize/optimize
  - Want to verify that synthesis/transforms did not introduce an error

Question

- Given a state machine with N states:
  - How long of an input sequence do I need to visit all N states?

Cornerstone Result

- Given two FSM’s, can test their equivalence in finite time
- N.B.:
  - Can visit all states in a FSM with finite input strings
    - No longer than number of states
    - Any string longer must have visited some state more than once (by pigeon-hole principle)
    - Cannot distinguish any prefix longer than number of states from some shorter prefix which eliminates cycle (pumping lemma)

FSM Equivalence

- Given same sequence of inputs
  - Returns same sequence of outputs
- Observation means can reason about finite sequence prefixes and extend to infinite sequences which FSMs are defined over
Equivalence

- Brute Force:
  - Generate all strings of length |state|
  - (for larger FSM = the one with the most states)
  - Feed to both FSMs with these strings
  - Observe any differences?
- How many such strings?
  - |Alphabet|^|states|

Smarter

- Create composite FSM
  - Start with both FSMs
  - Connect common inputs together (Feed both FSMs)
  - XOR together outputs of two FSMs
    - Xor’s will be 1 if they disagree, 0 otherwise
  - Ask if the new machine ever generate a 1 on an xor output (signal disagreement)
    - Any 1 is a proof of non-equivalence
    - Never produce a 1 \(\rightarrow\) equivalent

Creating Composite FSM

- Assume know start state for each FSM
- Each state in composite is labeled by the pair \(\{S_1, S_2\}\)
  - How many such states?
  - Compare to number of strings of length #states?
- Start in \(\{S_1_0, S_2_0\}\)
- For each symbol \(a\), create a new edge:
  - \(T(a,\{S_1_0, S_2_0\}) \rightarrow \{S_1_i, S_2_j\}\)
    - If \(T_1(a, S_1_0) \rightarrow S_1\) and \(T_2(a, S_2_0) \rightarrow S_2\)
- Repeat for each composite state reached

Composite DFA

- How much work?
  At most \(|alphabet|^{|State1|*|State2|} \text{ edges} = \text{ work}
- Can group together original edges
  - i.e. in each state compute intersections of outgoing edges
  - Really at most \(|E_1|^{|E_2|}

Non-Equivalence

- State \(\{S_1, S_2\}\) demonstrates non-equivalence iff
  - \(\{S_1, S_2\}\) reachable
  - On some input, State \(S_1\) and \(S_2\) produce different outputs
- If \(S_1\) and \(S_2\) have the same outputs for all composite states, it is impossible to distinguish the machines
  - They are equivalent
- A reachable state with differing outputs
  - Implies the machines are not identical

Empty Language

- Now that we have a composite state machine, with this construction
- **Question**: does this composite state machine ever produce a 1?
  - Is there a reachable state that has differing outputs?
Answering Empty Language

- Start at composite start state \( \{S_1^0, S_2^0\} \)
- Search for path to a differing state
- Use any search (BFS, DFS)
- End when find differing state
  - Not equivalent
- OR when have explored entire reachable graph w/out finding
  - Are equivalent

Reachability Search

- Worst: explore all edges at most once
  - \( O(|E|)=O(|E_1|^*|E_2|) \)
- When we know the start states, we can combine composition construction and search
  - \( i.e. \) only follow edges which fill-in as search
  - (way described)

Creating Composite FSM

- Assume know start state for each FSM
- Each state in composite is labeled by the pair \( \{S_1, S_2\} \)
- Start in \( \{S_1^0, S_2^0\} \)
- For each symbol \( a \), create a new edge:
  - \( T(a, \{S_1^0, S_2^0\}) \rightarrow \{S_1, S_2\} \)
  - \( T_1(a, S_1^0) \rightarrow S_1, \text{and } T_2(a, S_2^0) \rightarrow S_2 \)
- Repeat for each composite state reached

Issues to Address

- Obtaining State Transition Graph from Logic
- Incompletely specified FSM?
- Know valid (possible) states?
- Know start state?
Getting STG from Logic

• Brute Force
  – For each state
    • For each input minterm
      – Simulate/compute output
      – Add edges
  – Compute set of states will transition to
• Smarter
  – Exploit cube grouping, search pruning
    • Cover sets of inputs together
  – Coming attraction: PODEM

Incomplete State Specification

• Add edge for unspecified transition to
  – Single, new, terminal state
• Reachability of this state may indicate problem
  – Actually, if both transition to this new state for same cases
    • Might say are equivalent
    • Just need to distinguish one machine in this state and other not

Valid States

• Composite state construction and reachability further show what’s reachable
• So, end up finding set of valid states
  – Not all possible states from state bits

Start State?

• Worst-case:
  – Try verifying for all possible start state pairs
  – Identify start state pairs that lead to equivalence
    • Candidate start pairs
• More likely have one (specification) where know start state
  – Only need to test with all possible start states for the other FSM

Summary

• Finite state means
  – Can test with finite input strings
• Composition
  – Turn it into a question about a single FSM
• Reachability
  – Allows us to use poly-time search on FSM to prove equivalence
    • Or find differentiating input sequence

Admin

• Assignment 6 due Monday
• Reading for Monday and Wednesday on blackboard
• Q for Monday’s lecture [Feedback]
  – Assuming I’m not going to give a hard assignment based on output encoding
  – Prefer:
    • Make two separate lectures?
    • Briefly touch on output cover (similar to what did)?
Big Ideas

• Equivalence
  – Same observable behavior
  – Internal implementation irrelevant
  • Number/organization of states, encoding of state bits…

• Exploit structure
  – Finite DFA … necessity of reconvergent paths
  – Structured Search – group together cubes
  – Limit to valid/reachable states

• Proving invariants vs. empirical verification