Design Verification

Lecture 19 - Error-Directed Simulation-Based Verification

1. Motivation:
    - Formal techniques may not be able to offer comprehensive verification across a variety of designs
    - Formal methods attempt to verify the correctness of a system by implicitly consider all possible behavior of the models that represent the system. On the other hand, simulation-based approaches can only consider a limited range of behaviors.
    - $\rightarrow$ Without using formal methods, can we still show that the design is correct with *good* confidence?
    - Key: simulate the design with some (perhaps large) number of input patterns, and check the output of the design against the expected behavior
    - Issues:
        - What patterns to apply the design with?
        - How many patterns to apply?
        - How to assess verification measure?
2. What patterns to apply the design with during verification

    $\rightarrow$ Simplest is to use (pseudo) random patterns.

    - Apply random patterns to both design and golden machine/model
    - Compare the outputs from design and the expected outputs from golden machine/model
    - If any mismatch, design has bugs
    - If no mismatch, how much confidence do we have that our design is bug-free?
    - Key: pseudo-random patterns are effective in catching easy bugs early in the design phase

    $\rightarrow$ To catch corner cases, use intelligent patterns or formal methods.

    - Derive patterns to cover cases that pseudo-random vecs will likely miss
      $\longmapsto$ This may increase the confidence we have in the patterns

- If all fails, model corner case as a property, and verify property

Example 1

3. How many patterns to apply

   → To answer this question, we need to have some measure of coverage each pattern exercises in the design.

   Possible coverage metrics include:

   - Number of gates experienced both logic 0 and logic 1
   - Number of gates experienced both logic 0 and logic 1, and its effect could be observed at the primary output(s)
   - Number of gates verified, if each gate could have been incorrectly modeled
   - Number of states traversed in the FSM
   - Number of design errors covered via error modeling
   - If the design is described at a higher level, such as VHDL, then coverage metrics include statement coverage, condition coverage, and path coverage, as in software-testing coverage metrics.
   - Number of mutants covered in mutation testing

   → These metrics give us a guide to when we should stop applying more patterns to the design.

4. Vector dependence of error

   - Unlike stuck-at faults, where faulty value is constant, the erroneous value of an error can change depending on the values applied

5. Collection of Error Models (Campenhout, et al., ACM Trans. DAES), 2000

   - Injection of errors is synonymous to idea of mutants. These Error help us to distinguish the erroneous machine from the error-free one.
   - But what types of errors ought we model?
   - Basic error models:
     - Bus SSL error (SSL): A bus of 1 or more lines (totally) stuck at 0 or 1.
     - Module substitution error (MSE): mistakenly replacing a module by another.
     - Bus order error (BOE): incorrectly ordering the bits in a bus.
     - Bus source error (BSE): connecting a module input to a wrong source.
     - Bus driver error (BDE): mistakenly driving a bus with two sources.
   - More complex error models:
     - Bus count error (BCE): defining a module with more or fewer input buses than required.
     - Module count error (MCE): incorrectly adding or removing a module.
     - State count error (SCE): incorrect finite state machine with an extra or missing state
     - Next state error (NSE): incorrect next state function in a FSM.
   - Conditional errors:
     - A conditional error (C, E) consists of a condition C and a basic error E: **interpretation:** error E is only active when C is satisfied.
     - Example: $(X_2 = 0, YSSL0)$
   - In general, BSE is most common error type.
   - These error models also apply to higher-level circuit descriptions, such as RTL.
   - One can generate pseudo-random patterns and see how many of each type of error has been detected. For the remaining errors, generate the pattern by an automatic test pattern generator (ATPG).
   - Assumption: single-error present at a time
   - $$Error\ Coverage = \frac{total\ \#\ errors\ detected}{total\ \#\ errors\ in\ circuit}$$

6. Automatic Test Generation for Design Errors

   → To detect an error, we must

   - excite the error: apply value such that the error-free and error values on the gate of interest differ
   - propagate error effect: propagate effect of the excited error to a primary output (to make the error observable at the PO's).

   Example 2

   Example 3

   Example 4

7. Vectors to detect design errors

   - the number of vectors necessary to detect all stuck faults in an n-input gate is n+1 (considering only equivalent faults)
   - the number of vectors necessary to detect gate substitution is smaller

   Example 5

8. Classification of vectors (C-sets)

   - V_{null} = vector with all 0's
   - V_{all} = vector with all 1's
   - V_{odd} = set of vectors with odd number of 1's
   - V_{even} = set of vectors with non-zero, even number of 1's
   - Thus, for multi-input gates, V_{odd} contains at least 2 vectors
     for 2-input gates, V_{even} is empty

   Example 6

9. Testing design errors with C-sets
   - single-input gate substitution error detectable by V_{null} or V_{all}
   - multi-input gate substitution error detectable by 3 vectors: (1) V_{null}, (2) one vector from V_{odd}, (3) V_{all} if even number of inputs OR one vector from V_{even} if odd number of inputs

   Example 7

   Example 8

10. For gate-count errors
    - extra gate: detectable if complete test set for gate substitution errors available

- missing gate: need to model as a gate-substitution problem

11. Many stuck-at ATPG algorithms available, is it possible to model design error ATPG as stuck-at ATPG?

    - Need: map design error into a *gate-replacement module* (may involve multiple gates)
    - Assign a set of stuck-at faults F correspond to the new ckt
      $\longmapsto$ essentially, these faults force the C-set applicability
    - Call stuck-at ATPG to target F, the test set for F can also capture the original design error

    Example 9

12. Overall algorithm

    for each gate g
      for each design error involving g
        replace g with the corresponding replacement module
        assign the corresponding stuck faults in the replacement module
        call stuck-at ATPG, and store vectors generated

13. Sequential circuits

    - Similar approach, but need time-frame expansion during excitation and propagation

14. Property-directed simulation-based verification (symbolic simulation)
    - Instead of targeting errors, target properties
    - Want: simulate all possible input patterns to make sure property is held
      - normal logic simulation simulates 1 vector at a time (or parallel-pattern of k vectors)
      - instead of simulating logic values, simulate symbols such that all possible input patterns are accounted for
    - In combinational circuits, similar to equivalence checking
    - In sequential circuits, similar to symbolic trajectory evaluation

    Example 10

15. Symbolic Trajectory Evaluation (STE) for sequential circuits
    - Given a property that spans several time frames, check if it is held in the implementation circuit
    - Again, symbolic simulation is used, time-frame by time-frame
      $\longmapsto$ similar to image computation, except finite length and specific starting state and input constraints

    Example 11

16. Can we reduce the number of symbolic variables? Yes, by parameterization.
    - map variables to a set of parameters
    - may reduce some variables to constants if a test-bench is provided

    Example 12

    Example 13

17. Modeling of unknowns in STE via dual-rail encoding

    - (1 0) = logic 0
    - (0 1) = logic 1
    - (0 0) = logic X
    - Use parameters to represent a set of 3-valued vectors that covers the entire input space

    Example 14