Concurrent Fault Simulation for Structural Verilog code

Aakarsh A, Department of E&ECE, IIT Kharagpur

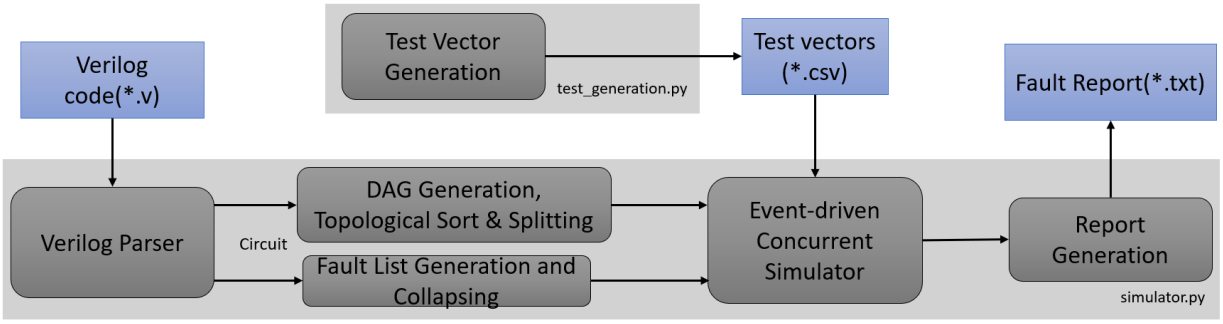
1 Introduction and Functionality

This final iteration extends the concurrent single-stuck-at-fault simulator to support a more realistic digital-test flow. The tool still takes as input a Verilog module (*.v) and a test vector file (*.csv) and produces a text report with fault coverage, listing which single stuck-at faults are detected and which remain undetected.
Compared to the earlier version, the new design:

1. Supports a larger primitive set: combinational gates and, or, not, nand, nor, xor, xnor and sequential dff instances, all with variadic fan-in (e.g. nand (y, a, b, c);).
2. Implements fault collapsing at two levels: NOT-chain collapsing for single-fanout inverter chains, Dominance/equivalence collapsing for AND/OR/NAND/NOR families, reducing redundant faults using standard dominance rules.
3. Uses an event-driven, bit-parallel simulation kernel instead of recomputing every gate for every vector, which significantly reduces work when only a few nets change between vectors.
4. Adds a separate vector generator script, which parses the top module's ports and automatically generates exhaustive test vectors, including clock-aware patterns to create rising edges for DFFs.

The **code is uploaded to GitHub** with link attached to the title & instructions to run are in **README**.

2 Architectural View and Description



2.1 Verilog Parser and Vector Generator

2.1.1 Verilog Parser

The *parse verilog structural* function reads the Verilog source, strips line and block comments, and then extracts modules using a simple regex form.

- For the selected top module, it collects primary inputs and outputs from both: legacy body declarations (input A,B; output Y;), and ANSI-style port lists (module m(input A, B, output Y);).

- Records all internal wires via wire declarations.
- Finds primitive gate instances for all supported primitives: and, or, not, nand, nor, xor, xnor.
- For each instance, it creates a Gate object: The code allows variadic inputs for logic gates and special handling for DFF ((D,clk,Q)).
- All gates, nets, primary inputs/outputs and DFFs are recorded into a *Circuit* object. The Circuit.finalize() method then builds: *drivers[net]* – which gate drives each net. *fanout[net]* – list of nets that depend on a given net (for event-driven propagation). *statenodes* and *dffdefs* – treating DFF Q outputs as state.

2.1.2 Vector generator

The separate generatevectors.py file reuses a tiny port parser to discover the top module's input ports. Given the list of inputs, it:

- Enumerates all 2^N combinations over non-clock inputs.
- If a clock input (e.g. clk) exists, it emits two rows per combination: first with clk = 0, then second with clk = 1, producing a 0 to 1 rising edge for the DFFs in the main simulator.
- Writes the resulting patterns as a CSV file with a header row matching the port names.

This script directly automates the "Test vector generation" step for arbitrary small structural HDL modules. Input ports less than 20.

2.2 Levelization, Fanout Graph and Event-Driven Structures

After parsing the Verilog netlist, the tool organises the circuit into a Circuit object that stores gates, nets, primary inputs/outputs and DFFs. To safely evaluate combination logic, it computes a topological order using topo_order(): DFFs are treated as cut points, only combinational gates are considered, and a Kahn-style algorithm orders gates so that inputs are always available before their outputs are computed. In parallel, the simulator builds a fanout map net_to_gates, which records, for each net, the list of gates that read it. This combination of topological ordering and fanout information is what enables the **event-driven kernel**. When a PI or DFF output changes, only the gates fed by that net are enqueued, and their changes propagate forward through the fanout graph. As a result, the simulator no longer recomputes every gate for every vector; instead, it incrementally updates only those regions of the circuit that are actually affected by input or state changes.

2.3 Fault List Generation, NOT-Chain Collapsing and Dominance Collapsing

The simulator starts with a raw single stuck-at fault list by attaching sa-0 and sa-1 faults to every net: primary inputs, internal wires, primary outputs and DFF Q outputs. This is complete but redundant, so two collapsing steps are applied. First, *NOT-chain collapsing* detects single-fanout inverter chains (e.g. A to NOT to B to NOT to C). Any fault on intermediate nodes like B or C is mapped back to the source net (A) with inverted stuck-at polarity, removing redundant inverter faults while preserving behaviour. Second, *dominance/equivalence collapsing* is applied to multi-input AND, OR, NAND and NOR gates in fanout-free regions. For an AND, the tool keeps the output sa-1 fault and a single representative input sa-0 fault, and drops dominated cases such as output sa-0 and equivalent input sa-1 faults; analogous rules are used for OR, NAND and NOR with appropriate polarity changes.

XOR, XNOR, NOT and DFF faults are *not changed* by this dominance step (NOT is already handled by the chain collapse). Finally, the kept *(net, sa)* pairs are rebuilt into a compact fault list with new indices. This two-stage collapsing substantially reduces the number of faults that must be simulated, improving efficiency without losing essential coverage information

2.4 Event-Driven Concurrent Simulation

The main simulation loop is implemented in simulate_event_driven. It combines:

- **Bit-parallel encoding and batching (Bit parallel simulation):** The simulator runs faults in batches, where each batch encodes up to batch_bits faults plus one golden circuit in a single integer bit-vector. Bit 0 always represents the fault-free circuit, and bits 1. . . k correspond to different single stuck-at faults. For each net, two injection masks are pre-computed: one forcing selected bits to 0 (sa-0) and one forcing selected bits to 1 (sa-1). During simulation, when a net's value is computed, these masks are applied to overwrite the corresponding bits, so the good and all faulty circuits evolve concurrently in parallel.

- **Batch cache and event seeding:** For each fault batch, the simulator keeps a cache of the previous bit-vectors of all nets. When a new test vector arrives, it loads this cache and first updates only the primary input nets and DFF outputs (state nodes) according to the new vector and current DFF state, again applying the fault masks. Any PI or Q whose bit-vector changes compared to the cache is recorded as a changed net, and the gates that read those nets are added to an event queue. If there is no cache yet (first vector for that batch), the simulator simply evaluates the whole combinational cone once and then stores the result for future vectors.

- **Event-driven propagation and sequential update:** The event queue is processed in a topologically consistent order: whenever a gate is dequeued, its output bit-vector is recomputed using the current values of its inputs, masks are applied at its output net, and the result is compared with the old value. Only if the output actually changes are its fanout gates enqueued, so activity remains local to the part of the circuit affected by the new vector. On a clock rising edge, DFF Q outputs are updated from their D inputs (in bit-parallel form), and any Q that changes is treated like another changed net whose fanout cone is re-propagated. This way, both combinational changes and state transitions are handled in the same event-driven framework.

- **Detection, fault dropping and reporting:** After propagation settles for a vector, the simulator looks at each primary output's bit-vector and compares it to the golden value replicated across all bits. Any mismatch in bits 1. . . k indicates that the corresponding faults are detected by this test vector. These detected bits are decoded back to fault indices, the first detecting vector index is stored, and the faults are dropped from the undetected set so they are not simulated in later vectors. Once all vectors finish or all faults are dropped, the tool prints a coverage report summarising the number of faults after collapsing, how many were detected, the overall coverage percentage, and detailed lists of detected and undetected faults.

3 Simulation Results

```
Fault Simulation Report
========================
Top module       : something
Primary Inputs   : A, B, Sel
Primary Outputs  : y
Vectors simulated: 8
Faults (total)   : 10
Detected         : 10
Coverage         : 100.00%

Detected Faults (net, sa, first_detected_at_vector):
  id=   0 : A s-a-1 @ v3
  id=   1 : B s-a-0 @ v3
  id=   2 : B s-a-1 @ v1
  id=   3 : Sel s-a-1 @ v1
  id=   4 : w1 s-a-0 @ v1
  id=   5 : w2 s-a-0 @ v2
  id=   6 : w3 s-a-0 @ v1
  id=   7 : w3 s-a-1 @ v2
  id=   8 : y s-a-0 @ v2
  id=   9 : y s-a-1 @ v1

Undetected Faults:
```

```
Fault Simulation Report
========================
Top module       : seq_ckt
Primary Inputs   : clk, rst, d
Primary Outputs  : q, qbar
State Nodes (Q)  : q
Vectors simulated: 8
Faults (total)   : 10
Detected         : 8
Coverage         : 80.00%

Detected Faults (net, sa, first_detected_at_vector):
  id=   2 : d s-a-0 @ v4
  id=   3 : d_masked s-a-1 @ v2
  id=   4 : q s-a-0 @ v4
  id=   5 : q s-a-1 @ v1
  id=   6 : qbar s-a-0 @ v1
  id=   7 : qbar s-a-1 @ v4
  id=   8 : rst s-a-0 @ v8
  id=   9 : rst s-a-1 @ v4

Undetected Faults:
  id=   0 : clk s-a-0
  id=   1 : clk s-a-1
```