Wet 2 - Dry Section Answers

Submitted by

Amir Zuabi - 212606222

Alexey Vasilayev - 323686683

Basic Code Structure and High-Level Algorithm

1. Parse command-line arguments to obtain "specification" and "implementation" top module names.
2. Parse Verilog files into HCM designs.
3. Flatten both top-level designs.
4. Create a SAT variable for every node in each flattened design.
5. Encode all logic gates into CNF using Tseytin transformation.
6. Match primary inputs by name and add equivalence constraints.
7. Build a miter circuit by XOR-ing matching outputs and DFF D signals.
8. OR all XOR results together and assert that at least one mismatch exists.
9. Run the SAT solver to check equivalence (and print a CEX on SAT).
10. Export the generated CNF in DIMACS format.

Exact Explanation of the Comparison Algorithm

A single combined SAT instance is constructed that contains both the specification and the implementation circuits. For every logic gate in both designs, a fresh SAT variable is introduced to represent the gate's output, and CNF clauses are added using "Tseytin" encoding to enforce the correct Boolean behavior.

Primary inputs are matched by signal name and constrained to be logically equivalent, ensuring that both circuits are evaluated under the same input assignment. Flip-flops are also matched by name: the Q outputs of corresponding DFFs are tied together to represent identical states, while their D inputs are added to the set of signals to be compared.

For each matched primary output and each matched DFF D signal, an XOR node is created to indicate a functional mismatch between the specification and the implementation. All XOR results are then OR-ed together into a single miter output, and a clause is added to force this miter output to be true. If the SAT solver returns SAT, a counterexample input vector exists that exposes a mismatch. If the solver returns UNSAT, no such mismatch exists and the two circuits are functionally equivalent.

Handling of Constants

- Global constant nodes VDD and VSS are modeled explicitly in the SAT instance.
- The SAT variable corresponding to VDD is forced to true (1) using a unit clause.
- The SAT variable corresponding to VSS is forced to false (0) using a unit clause.
- This guarantees correct behavior for constant-driven logic throughout the design (no false SATs).

Support for XOR Gates and Tseytin Encoding

XOR gates are supported by explicitly encoding their Boolean behavior using Tseytin transformation. For each XOR gate, a fresh SAT variable is introduced to represent the XOR output.

Given an XOR gate with inputs a and b, and output z, the constraint $z = a \oplus b$ is enforced using the following four CNF clauses:

- $(\neg a \vee \neg b \vee \neg z)$
- $(a \vee b \vee \neg z)$
- $(a \vee \neg b \vee z)$
- $(\neg a \vee b \vee z)$

These clauses ensure that z is true if and only if a and b have different logical values, and false otherwise. This encoding is used both for modeling explicit XOR gates in the circuit and for building the miter, where XOR nodes are created to detect mismatches between corresponding outputs of the specification and implementation circuits.