Formal Verification: theory and practice

Vigyan Singhal
Oski Technology, Inc.
Fremont, CA, USA
Agenda

- Design and verification flow
- Theory
  - Model checking use model
  - Proof engine algorithms
  - Abstraction refinement algorithms
- Practice
  - Formal testplanning
  - Verification strategies
  - Links to coverage
System-on-Chip (SoC) design, verification

- **Design**: Specification to micro-architecture and down to implementation
- **Verification**: block-level (scheduler) to unit-level (USB interface unit) to chip-level (Mobile applications processor)
- Verification consumes about 70% of resources
- Each silicon re-spin costs more than $2M
- Most frequent cause of re-spin: functional bugs
Verification challenge

Bottlenecks in Design Cycles:
Survey of 545 engineers by EETIMES 2000
Simulation-based verification

- BFMs convert transactions (Read, Write, ...) to bit-level interface (req, ack, rdata[63:0], wait, ...)
- Tests responsible for input generation, and for covering all the interesting cases
- Passive checker is checks end-to-end correctness (higher-level than design, not always synthesizable)
Why formal verification?

- Traditional method (testing & simulation)
  - Design Under Test
  - Test vectors
  - Patterned vectors
  - Test properties
  - Deadlock...
  - Test results
  - Find counter-example

- Formal Verification
  - System Model
  - True or False
  - Equivalent to simulating all cases in simulation
  - No bug (according to the property)
  - Property checking (model checking) and Equivalence checking

- Cannot cover all possible cases
- Possibility of surviving subtle (corner case) bugs
Does the design have a given desirable property?

- Properties - Specification
- Constraints - Assumptions needed to prove Properties
- Design under test - Implementation

Verified or CounterExample

\[ p \]

\[ q \]
Under-the-hood...

- Evaluation model is cycle-level
  - Two stage evaluation
    - 0-delay combinational logic gates
    - 1-delay flip-flops
  - Synchronous front-end
    - Design synthesized to gates+flops
      - But not optimized
    - Properties, constraints also synthesized
      - OVL is easy
      - PSL, SVA is non-trivial
Is there a sequence of input assignments such that \( p \) is 1 at any finite time?

Combinational gates + flops (with initial values)
Model checking

Is there a sequence of input assignments such that $p$ is 1 at any finite time time?

Combinational gates + flops (with initial values)
Complexity: PSPACE-hard (Aziz 93)
Key notion: “reachable states”

reachable states: the set of states reachable (under any input sequence) from the input state

The behavior of the design from unreachable states is not relevant to the correctness of the property

Finding the set of reachable states is hard for many designs (more than 100 registers => more than $2^{100}$ possible states!)

=> state space explosion problem!
Problem complexity is coarsely related to number of flops in the cone of influence of the property.
Reachability Example

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2'b00;
  else case( st )
    2'b00: if (~a) st <= 2'b01;
    2'b01: st <= 2'b10;
    2'b10: if (a) st <= 2'b00;
  endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1'b0;
  else if (~a | b) b <= 1'b0;
  else b <= 1'b1;
```

Prove (st == 2'b01) => ~b
Reachable state computation: explicit

1. Reached set = \{000\}
   State to explore = 000
2. Reached set = \{000, 001, 010\}
   State to explore = 001
3. Reached set = \{000, 001, 010\}
   State to explore = 010
4. Reached set = \{000, 001, 010, 100, 101\}
   State to explore = \{100, 101\}
5. Reached set = \{000, 001, 010, 100, 101\}
   State to explore = \{101\}
6. Reached set = \{000, 001, 010, 100, 101\}

Num of steps = Num of reached states
Reachable state computation: symbolic

1. Reached set = \{000\}
   \[= (~st[1] \& ~st[0] \& ~b)\]

2. Reached set = \{000, 001, 010\}
   \[= (~st[1] \& (~st[0] \& b \mid st[0] \& ~b))\]

3. Reached set = \{000, 001, 010, 100, 101\}
   \[= (~st[0] \mid ~st[1] \& ~b)\]

**Num of steps = “diameter” of circuit**

For many circuits, diameter << number of reachable states
- except, circuits with counters/timers, or other some counting-like control
Reachability: forward, backward

- Start from $I_0$ ($Z$) = set of initial (bad) states
- Perform forward (backward) reachability analysis to compute a reached state set $R$ ($B$)
- Determine if $R$ ($B$) intersects $Z$ ($S_0$)
Background: BDDs (binary decision diagrams)

\[ \text{out} = (a \& b) | c \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Ordered Decision Tree
Background: BDDs

- **Reduced Ordered BDD**
  - Merge isomorphic nodes
  - Remove redundant tests

- BDD size dependent on variable order
Background: Operations on BDDs

The most important for model checking:

- **Boolean operations**: and, not, or
- **Quantification**: \( \exists x \ f(x,y) \)
- **Specialized operators**
  - Generalized cofactor (constrain)
  - Restrict
  - Compatible projection
  - etc.

See (Somenzi 99) for more details
Reachability computation

Transition (transfer) functions (TF):

\[ y_1 = \neg(x_1 \lor x_2) \land w \]
\[ y_2 = x_1 \land w \]

Initial state \( R_0 = \neg x_1 \land \neg x_2 \)
Forward Reachability: with BDDs

R0 = Initial Set = \{00\} = \sim x_1 \& \sim x_2
R1 = \{00, 10\} = \sim x_2
R2 = R^* = \{00, 10, 01\} = \sim x_1 \mid \sim x_2
Symbolic reachability algorithm

Reachability algorithm:

\[ R_{next} = I_0 \]

\[ \text{do} \{ \]

\[ R = R_{next} \]

\[ R_{new} = \text{image}(R, TF) \]

\[ R_{next} = R + R_{new} \]

\[ \text{until} \ (R == R_{next}) \]

Image computation, \( \text{image}(R, TF) \):

\[ \text{image}(Y) = \exists (w, X) \ [ R(x_1, x_2) \ & \ (y_1 = (~x_1 | x_2) \ & \ w)) \ & \ (y_2 = (x_1 \ & \ w))] \]

\[ \text{image}(X) = \text{image}(Y)_{Y < X} \]

\[ R^* = \sim x_1 \ | \ \sim x_2 = \{00, 01, 10\} \]
SAT-based model checking

- **Bounded MC**
  - Incomplete: can only find bugs, no proofs
  - [Biere 99]

- **Unbounded MC**
  - Complete, used for proofs
  - Induction-based [Sheeren 00]
  - Backward reachability [McMillan 02, Ganai 04]
  - Interpolants [McMillan03]
Background: SAT Problem definition

Given a CNF formula, $f$:

- A set of variables, $V$ $(a,b,c,d)$
- Conjunction of clauses CNF $(C_1, C_2, C_3)$
- Each clause: disjunction of literals over $V$

Does there exist an assignment of Boolean values to the variables, $V$ which sets at least one literal in each clause to ‘1’?

Example: $(a + b + \overline{c})(\overline{a} + c)(a + \overline{b} + c)
\begin{align*}
C_1 & \quad C_2 & \quad C_3 \\
& a = 1 & \quad b = 0 & \quad c = 1
\end{align*}$
Background: combinational SAT

Can circuit output “p” be 1?

\[(a \lor \neg g) \land (b \lor \neg g) \land (\neg a \lor \neg b \lor g)\]

CNF(p)

\[(-g \lor p) \land (\neg c \lor p) \land (g \lor c \lor \neg p)\]

\[ CNF(p) \land p \]

p is satisfiable when the formula CNF(p) \land p is satisfiable

Complexity: NP-complete (Cook 71)
Background: Basic SAT algorithm

1. If \( A = \emptyset \), proceed.
2. Check if the clause is empty.
   - If yes, \( A \) is UNSAT.
   - If no, proceed.
3. Check for conflict.
   - If yes, deduce conflict clause and backtrack.
   - If no, continue.
4. Check if \( A \) is total.
   - If yes, \( A \) is SAT.
   - If no, branch: add some literal to \( A \).
SAT-based model checking

- **Bounded MC**
  - Incomplete: can only find bugs, no proofs
  - [Biere 99]

- **Unbounded MC**
  - Complete, used for proofs
  - Induction-based [Sheeren 00]
  - Backward reachability [McMillan 02, Ganai 04]
  - Interpolants [McMillan03]

Complexity: PSPACE-complete (Aziz 93)
Bounded Model Checking (Biere 99)

- Unfold the model \( k \) times:
  \[
  U_k = C_0 \land C_1 \land ... \land C_{k-1}
  \]

- Use SAT solver to check satisfiability of
  \[
  I_0 \land U_k \land Z_k
  \]
  - A satisfying assignment is a counterexample of \( k \) steps
Induction-based (simple case)

- Base step (initial state is not a bad state)
  - \( I_0 \land Z_0 \) is unsatisfiable

- Induction step: (if \( n \)-th state is not bad, then \( (n+1) \)-th states are not bad)
  - \( \neg Z_n \land U_n \land Z_{n+1} \) is unsatisfiable
  - (i.e. \( \neg Z_n \Rightarrow \neg Z_{n+1} \))

- If Base and Induction are each unsatisfiable, property is true
- Else, need to increase induction step (next slide)
Reachable state computation: symbolic

1. Reached set = \{000\}
   \[\sim st[1] \land \sim st[0] \land \sim b\]

2. Reached set = \{000, 001, 010\}
   \[\sim st[1] \land (\sim st[0] \land b \lor st[0] \land \sim b)\]

3. Reached set = \{000, 001, 010, 100, 101\}
   \[(\sim st[0] \land \sim st[1] \land \sim b)\]

Num of steps = “diameter” of circuit

For many circuits, diameter \ll number of reachable states
- except, circuits with counters/timers, or other some counting-like control
Induction-based (simple case)

- Simple induction may be too simple
  - Require $Z$ to be true for more than 1 cycle

Solution:
  - Require $Z$ to be true for more than 1 cycle
Induction-based (k-case)

- Recall: \( U_k = C_0 \land C_1 \land \ldots \land C_{k-1} \)
- Two formulas to check:
  - Base case: \( I_0 \land U_k \land Z_k \) is unsatisfiable
  - Induction step: \( \neg Z_0 \land U_k \land Z_k \) is unsatisfiable
  - (i.e. \( \neg Z_0 \implies \neg Z_k \))

If both are valid, then \( \neg Z \) always holds
- If Base is satisfiable, found a counter-example
- Else, increase \( k \) and try again
Unfortunately, k-induction is not complete
- Some properties not k-inductive for any k

Simple path restriction:
- There is a path to $\neg Z$ iff there is a *simple* path to $\neg Z$ (path with no repeated states).
Unique paths Induction (Sheeran 99)
- Restrict unrolled circuit to be loop-free
- Unrolling upto *recurrence diameter*
- i.e. length of longest loop-free path

Stronger induction methods can be expensive

*But,* some properties very easy to prove by induction, very hard by state-space traversal

Simple induction can be cheap, first step in verification flow
Backward reachability: symbolic

Backward Reachability algorithm:
Bnext = Z
do {
    B = Bnext
    Bnew = pre-image(B, TF)
    Bnext = B + Bnew
} until (B == Bnext)

- SAT clauses used to represent backward reachable sets
- SAT used for
  - pre-image
  - union (+)
  - fixed-point check
Interpolants (approximate forward reachability)

Interpolant algorithm:

\[ k = 0 \]

do forever {
    Do k-step BMC from \( I_0 \); if Z reachable, done cex
    \( R = I_0 \)
    while R does not intersect Z do {
        Rnew = over-image( R, TF, k )
        If Rnew is contained in R, done proof
        R = R + Rnew
    }
}

- Successive inner-loop approximations are more accurate (as k increases)
- If we reach Z in inner loop, may be a spurious path
Observations

- **BDD-based**
  - Most robust (predictable)
  - Upto 200-400 flops; exceptions for specific circuit types

- **SAT – bounded model checking**
  - Best at finding quick bugs
  - No proofs

- **SAT – induction**
  - Can quick proofs for specific properties

- **SAT – backward reachability**
  - Sometimes scales to larger examples than BDDs
  - May get stuck in enumeration of backward reach set

- **SAT – interpolation**
  - Works magically sometimes
  - Characterization not known
Open problems

- Exploit synergies between various engines
  - Generate invariants through induction
  - Backward reachability can enlarge bad state target
  - Use BDD-based reachability as back-up for enumeration in SAT-based reachability
  - For interpolation, use spurious path to refine approximation

- Characterize behavior of interpolation

- A general-purpose model checker built on top of existing model checking algorithms

- Upcoming model checking competition (à la SAT Competition)
Abstraction

- Construct $C'$ such that $C \leq C'$
  - e.g. a sub-circuit $C'$ (e.g. localization)

- If $C'$ satisfies property $\phi$, then $M$ satisfies

- If $C'$ does not satisfy $\phi$, then can’t say anything

- Localization abstraction (Kurshan 89)
- Predicate abstraction (Graf 97)
  - software, infinite-state systems
Model Checking & Abstraction

- Problem: Abstraction too coarse
  - Solution: Refine abstraction
- Make boxes smaller

Init

BAD STATES
Model Checking & Abstraction

- Problem: Abstraction too coarse
  - Solution: Refine abstraction
    - Make boxes smaller
Localization (Kurshan 89)
Localization (Kurshan 89)

Localization is a valid abstraction
Abstraction refinement

Build abstract model

Model check abstract model

Refinement

Check abstract counterexample on concrete model

True on concrete model

False on concrete model

Infeasible

spurious counterexample

real counterexample

true

Abort

Infeasible
History

- 1980: [Clarke, CMU] Theory for formal verification for parallel programs
- 1990: [Burch, CMU] Academic implementation of formal verification for hardware designs, using BDDs
- 1995: First commercial formal verification: AT&T FormalCheck, RuleBase
- 1999: [Sheeran, Chalmers] SAT Induction-based model checking
- 2001: [Moskewicz, Princeton] Fast SAT solvers
- 2002: [Chauhan, CMU] Automatic abstraction refinement, commercialized in Synopsys Magellan
- 2003: [McMillan, Cadence] SAT Interpolant-based model checking
- 2003: Formal verification for block-level: Jasper, Synopsys Magellan
- 2005: Assertion/property languages standardized: PSL, SVA
Commercial practice

- First-gen tools (1995): AT&T FormalCheck and IBM RuleBase
  - Needed significant user expertise, required design modification

  - Averant, Real Intent
    - Mostly used for very local pre-defined checks
  - 0-In, Magellan
    - Hybrid, combined with simulation
    - Bug-finding, not full coverage

- Third-gen (2003 -)
  - Jasper, Cadence ISV
  - Can be used for block-level verification: designer-sized block
  - Require users to break proofs, and apply dedicated resources
Commercial practice

- Usage models
  - Full-chip usage
    - Asynchronous clock-domain crossing
    - Lint checking
    - Automatic
  - Bug finding
    - Hybrid tools
    - Some hand-holding
  - Full proofs
    - Critical blocks or functionality
    - Dedicated use

- Organization
  - Dedicated formal groups for full proofs
    - NVIDIA, Sun, Sony, …
Where to apply formal

- Control, data transport: concurrency, multiple streams
- Arbiters of many kinds
- On-chip bus bridges
- Power management unit
- DMA controller
- Host bus interface unit
- Schedulers
- Interrupt controller
- Memory Controller
- Tag generator
- Credit manager block
- Standard interfaces (PCI Express, USB)
- Clock disable unit

Multiple, concurrent streams
Hard to completely verify using simulation
"10 bugs per 1000 gates"
-Ted Scardamalia, IBM
Where **not** to apply formal

- Data transform: often sequential, math operations
- Floating point unit
- Graphics shading unit
- Convolution unit in a DSP chip
- MPEG decoder
- Classification search algorithm

Single, sequential functional streams

“2 bugs per 1000 gates”

-Ted Scardamalia, IBM
Verification abstraction techniques

- Localization
  - ignore irrelevant logic
- Datapath abstraction
  - reduce datapath width
- Symmetry
- Sequence abstraction
  - Represent infinite sequence by two or three packets
- Memory abstraction
- Restriction
  - prove for a smaller state space
Example: AHB-AHB bus scheduler

- 32-bit datapath
- 16 different master agents
- History, policy-dependent
- AHB arbiter/scheduler
- Link-layer responses arbitrated with Master Packets
- Data Buffers store multiple packets (upto 32kb each)
Example: verification challenges

- **Simulation Complexity**
  - Timing relationship between the multiple masters
  - Error responses from the slave with arbitrary timing relationship to what is happening in the master

- **Corner-case**
  - Bridge locks up if slave issues error on 2\textsuperscript{nd} cycle of a 4-beat request from master A, and master B makes a new request on the same cycle the error is passed on to master A

- **Formal Complexity**
  - Data-width large
  - Supports upto 16 Master agents
  - Mathematical scheduling algorithm
  - Large data buffers
Property: for any master, sequence of Write packets arriving at DUT go out in the same order.

Scheduler algorithm, Link Layer responses, Reg Packets can all be arbitrarily chosen to prove this property.
Localization

- Free design logic that is irrelevant to property proof
- Increases legal space (removed logic is arbitrary)
- But decreases complexity

- Safe:
  - If removed logic is irrelevant, proof will go through
  - Else, trace (counterexample) shows relevance of logic

- Many tools allow user to perform localization
  - JasperGold, Synopsys Magellan, Cadence ISV
Property: for any master, sequence of Write packets arriving at Write Buffer go out in the same order

- Pick any arbitrary bit from the 32-bit datapath, say data[29]
- Prove correctness, one bit at a time
Datapath abstraction

- Have to assume, or prove, design control is independent of data bits
- To be complete, have to prove
  - For each bit
  - All bits travel together
- EDA tools may put it under the hood
Property: for any master, sequence of Write packets arriving at Write Buffer go out in the same order.

Add assumptions: “stopat data”; “assume (winner_vc == 3) \(\Rightarrow (data = data_m3)\)”
Assume-guarantee

- User commands may differ from tool to tool
- May require some modeling logic
- Decomposes total verification complexity
  - Remember, complexity is exponential
- To be complete,
  - Every assumption should be proven independently
Property: for any master, any (infinite) sequence of Write packets arriving into DUT go out in the same order.

Pick two arbitrary, consecutive packets. Mark them using a data bit. If DUT has a bug, it will show up in one of these two packets on the output.
**Memory abstraction - example**

- Track only the interesting packets (p1 and p2)
- Memory reduction: 10,000X here
  - Abstraction is independent of memory size
- Can be applicable to other applications, e.g. tag generator

```vhdl
always @(posedge clk)
  if (wr) && (data_in[73:72]==2'b01)
    p1addr <= addr;

  ...

  "assume (addr==p1addr) ➔
  (data_out[73:72]==2'b01)"
```

Inputs: wr, rd, addr[12:0], data_in[127:0]

Output: data_out[127:0]
Restriction

- Useful when the complexity is expected to be unmanageable
- Restrictions give partial proofs
- Restrict to
  - Certain modes: e.g. Memory transactions
  - Certain sequences: No back-to-back
  - Disallow exceptions: no interrupts
  - Get around known bugs: No Burst Write followed by Read
- Planned restrictions can help manage complexity, staged verification, schedule/resources
Open problems

- Implement abstraction refinement over proof engines
  - Allow arbitrary pairing of refinement algorithm with proof engine

- Automatic (or semi-automatic) methods for new abstraction
  - Possibly based on high-level design information: automatically extracted, or user-specified
Summary

- Formal verification is practical, but...
  - For well-selected problems
  - Requires user expertise
- Open problems in
  - Integration, and co-operation of proof engines
  - Automation in abstraction and abstraction-refinement methods
  - Use of design patterns
Functional verification closure

- **Track metrics**
  - Bugs, RTL changes, Number of tests

- **Coverage**
  - Structural (code)
    - Line, expression, FSM
    - 100% goal, style-dependant
  - Functional
    - Functional coverage points: input, output, internal, SM
    - Dependent on functional coverage points
    - For best results, define upfront in a coverage plan
      - else, can be biased by directed testplan, RTL bugs
Formal + Simulation

- Integrate simulation and formal
  - Tight integration of engine
  - Black-box integration
- Use of coverage targets as heuristic for goals
Synopsys Magellan methodology

- **Random simulation**: Deep narrow search
- **SAT-BMC**: Short-range exhaustive (wide) search (<10 steps)
- **Symbolic simulation / SAT-BMC**: Middle range exhaustive (wide) search (10-50 steps)
**Approach:** Test amplification

- Identify "seed states" reached through simulation (directed or random)
- Explore exhaustively behavior around seed states using formal methods
Functional verification closure
Functional coverage

- **Kinds**
  - Types of transactions: READ, WRITE, WRITEBURST, ...
  - Back-to-back transactions
  - Interface (handshake, backpressure, wait states, ...)
  - Interesting internal events
    - Control packets pass data packets
    - Design buffers are full
    - All Flow Control credits are used

- **All coverage points map to input space**
  - Directly (input points), or indirectly (output, internal)
Functional coverage: input space

- **legal space**
- **random tests (set 1)**
- **random tests (set 2)**
- **directed tests**
Functional coverage: test optimization

- Random constrained simulation
- Analyze, and re-bias mix
- Directed tests to cover holes
- Test optimization to improve regression time – days to hours
Coverage testplanning

- Early planning
  - Before start of verification (testbench development)
  - Enumerate early coverage points
  - Avoid risk of biasing by:
    - Design decisions
    - RTL bugs found
    - Directed simulation or formal testplan
  - Can add on coverage points any time
Coverage-driven formal flow

1. Specify constraints, properties, coverage points
2. Build BFM (synthesizable?)
3. Use formal to hit coverage
   1. Much easier, computationally, than property checking
4. Leverage simulation results to drive formal proof

- Synopsys: need to remove synthesizable BFM requirement
- Mentor: need to use coverage with formal/semi-formal
- Cadence: need to integrate simulation with formal
- Most promising unifier: SystemVerilog
  - Common verification kernel, framework for different technologies to cooperate
Formal verification is practical
- but, for well-selected problems...

Successful application requires
- Formal testplanning
- Dedicated resources
- Expertise for verification strategy

Watch out for commercialization of
- Formal methods for coverage closure
- Constraint specification
Where to apply formal

- Control, data transport: concurrency, multiple streams
  - Arbiters of many kinds
  - On-chip bus bridges
  - Power management unit
  - DMA controller
  - Host bus interface unit
  - Schedulers

- Interrupt controller
- Memory Controller
- Tag generator
- Credit manager block
- Standard interfaces (PCI Express, USB)
- Clock disable unit

Multiple, concurrent streams
Hard to completely verify using simulation
“10 bugs per 1000 gates”
-Ted Scardamalia, IBM
Where **not** to apply formal

- Data transform: often sequential, math operations
- Floating point unit
- Graphics shading unit
- Convolution unit in a DSP chip
- MPEG decoder
- Classification search algorithm

Single, sequential functional streams

"2 bugs per 1000 gates"

-Ted Scardamalia, IBM
Hierarchical Design/Verification Flow

Most of today's verification is performed at the chip level.

Minimal block-level verification.
0-in search: Tool flow

Instrumented Verilog RTL -> Standard Verilog Simulator -> Simulation Trace (Seed) -> 0-in Search 0-in Confirm

Verilog Tests and Testbench

Checker Control File

No firings in Verilog output

Standard Verilog Simulator

Firings Log

0-in View GUI

Standard Waveform Tool

Verilog simulator (Firing Replay)

Source: 0-in Design Automation Verification White-paper

© Oski Technology 2006