Semi-Formal Verification

Instructor: Dr. Hari Mony
Real Intent Inc.

EE 382M-11, Verification of Digital Systems
Spring 2020

Acknowledgements:
Dr. Jason Baumgartner
Dr. Jay Bhadra
Prof. Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

4/9/2020

Verification sign-off using Formal Verification

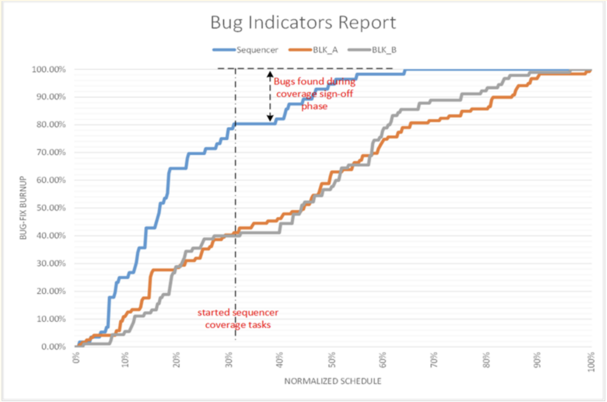
- FV has become mainstream verification methodology
  - Deployed in all major companies that develop complex SoCs
- FV has several advantages over dynamic simulation
  - Ability to generate exhaustive proofs
  - Ability to find corner-case bugs
  - Simpler testbench structure
  - Easy/Quicker to setup; Faster to discover bugs
- Can we do verification sign-off using FV?

Verification sign-off using Formal Verification

- To-enable signoff, we need a signoff metric
  - **Coverage** is the obvious answer
  - Common metric with dynamic simulation
- FV is supposed to cover 100% state-space, why is the need for **coverage?**
  - Unintentional over-constraints in FV environment
  - Design-complexity – cannot achieve exhaustive coverage leading to bounded proofs
- Are there advantages of doing verification sign-off using FV?

Advantages of Verification sign-off using FV?

- Bugs are found faster when using FV
- Debug cycles are 30% shorter on average
  - $>$50% of ASIC design project time is spent on verification



Acknowledgment: Hao Chen et.al, DVCON 2019

Verification sign-off using Formal Verification

- We still need to perform coverage closure for sign-off
- Is there a technology that can enable faster coverage closure?
  - **YES, Semi-formal Verification (SFV)**
  - Let's learn **SFV**

Outline

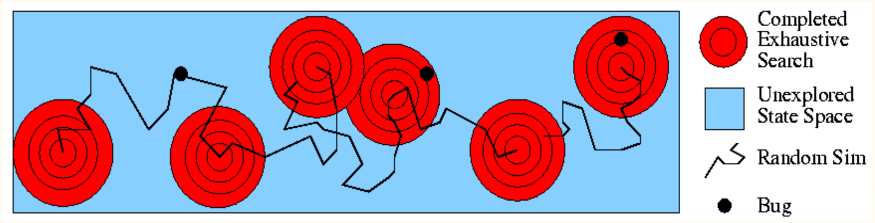
- Semi-formal verification (SFV) – What, Why, How?
- Brief overview of various semi-formal techniques
- Industrial SFV Experience
- Overview of IBM's RBSXS – FV tool with SFV technologies

Semi-formal Verification (SFV) – What, Why, How

- What is semi-formal verification (SFV)?
  - An attempt to combine the completeness of formal techniques with the speed, capacity and ease-of-use of simulation
  - Leverage formal techniques in a resource-bounded way
- Why are semi-formal techniques needed?
  - Critical for verification sign-off using FV
  - Corner-case bugs too complex for sim and too deep for formal
  - Critical for deep bugs
  - Key to scaling formal algorithms to large, complex designs
- How do semi-formal techniques work?
  - Augmenting simulation using formal techniques
  - Guiding simulation using formal techniques

SFV by Augmenting simulation

- Verification Problem: Check for *queue* overflow
  - FV techniques such as *bounded model checking (BMC)* cannot go deep enough
  - SIM has to get lucky – cannot consider all possible scenarios
  - BMC can do exh. search from a state such as *queue is 1/4 full*
  - Can Sim get the design to *queue is 1/4 full* state? **YES!**



Acknowledgment: J. Baumgartner, IBM

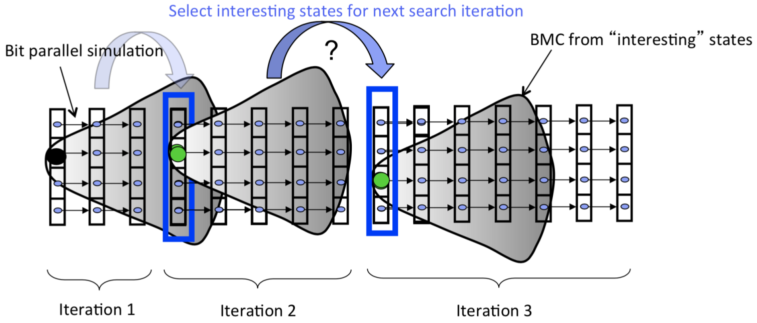
SFV by Augmenting Simulation

- Methodologically: Manual definition of *guideposts*
  - D. L. Dill and C. H. Yang. “Validation with guided search of state space”, DAC 1998.
  - Gorai et. al. “Directed-simulation assisted formal verification of serial protocol and bridge” DAC 2006.
  - Nadel et. al. “An Experience of Complex Design Validation: How to Make Semiformal Verification Work”, DVCON 2010.
  - Eslinger and Yeung. “Formal Bug Hunting with *River Fishing* Techniques”, DVCON 2019.
- Automatic or Tool Driven
  - Ganai et. al. “Siva: A system for coverage-directed state space search”, J. Electron. Test. 2001
  - IBM SFV tool *RuleBase-SixthSense*
  - Jasper-Gold (*Cycle-Swarm technique*)
  - Synopsys FV toolkit
  - Mentor Graphics FV toolkit

SFV by Augmenting Simulation: Challenges

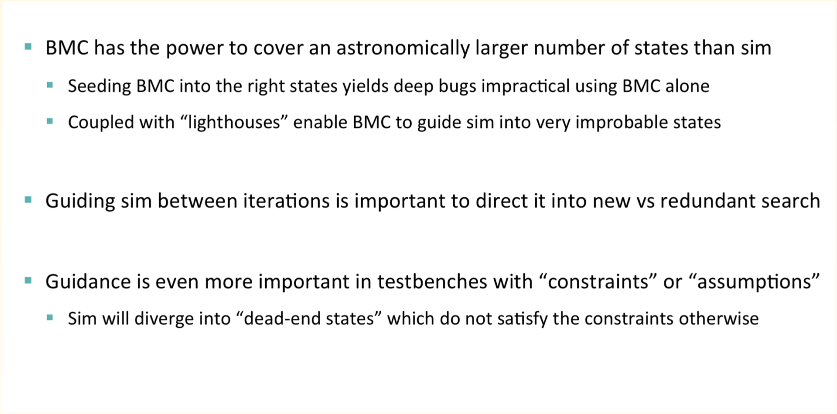
- Formal search is only effective if it is triggered near a fail
  - Cannot improve falsification capability otherwise
  - Techniques that make simulation "smarter" are applicable
    - e.g., better input pattern generation using biases
- Approaches
  - State prioritization: try to trigger iterations from new/interesting states
    - Apply rarity-analysis to find interesting states
    - M. K. Ganai and A. Aziz, "Rarity based guided state space search", GLSVLSI, ACM 2001
  - Light-houses/Stepping-stones: automated generation of *guideposts* towards fail
    - Unhit design states can be used as guideposts
    - Can use formal analysis to assert the lighthouses
    - Yalagandula et. al., "Automatic lighthouse generation for directed state space search", DATE 2000
  - *State-swarming technology in Jasper-Gold is essentially state-prioritization*

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)


Select interesting states for next search iteration
Bit parallel simulation
?
BMC from "interesting" states
Iteration 1
Iteration 2
Iteration 3

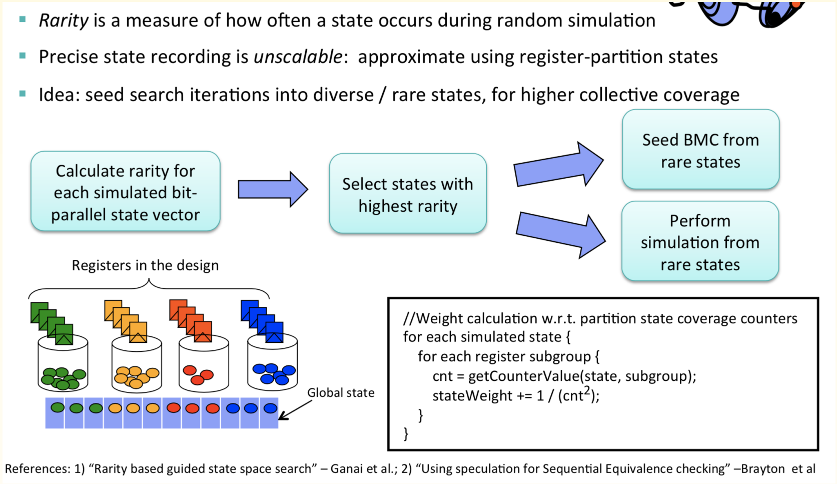
- Synergistic interaction of guided bit-parallel simulation and SAT-based bounded model checking (BMC)
- Each simulation word = 64 scalar bit-level simulations, evaluated in parallel (exploits data parallelism)
  - Often achieve highest throughput with 50-1000 words
- Many heuristics to select "interesting" states for later search iterations, to guide simulation + BMC

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)



- BMC has the power to cover an astronomically larger number of states than sim
  - Seeding BMC into the right states yields deep bugs impractical using BMC alone
  - Coupled with “lighthouses” enable BMC to guide sim into very improbable states
- Guiding sim between iterations is important to direct it into new vs redundant search
- Guidance is even more important in testbenches with “constraints” or “assumptions”
  - Sim will diverge into “dead-end states” which do not satisfy the constraints otherwise

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)



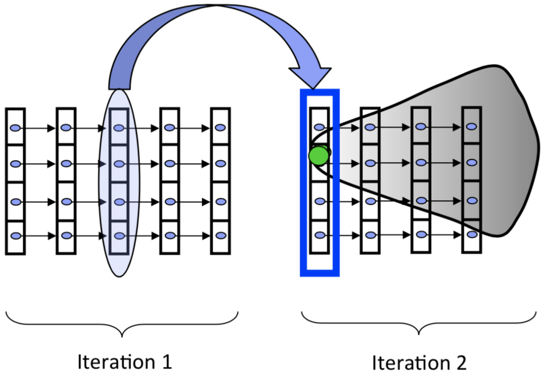
- *Rarity* is a measure of how often a state occurs during random simulation
- Precise state recording is *unscalable*: approximate using register-partition states
- Idea: seed search iterations into diverse / rare states, for higher collective coverage

$$\begin{tikzcd}
 & & \text{Seed BMC from rare states} \\
\text{Calculate rarity for each simulated bit-parallel state vector} \arrow[r] & \text{Select states with highest rarity} \arrow[ur] \arrow[dr] & \\
 & & \text{Perform simulation from rare states}
\end{tikzcd}$$

```
//Weight calculation w.r.t. partition state coverage counters
for each simulated state {
   for each register subgroup {
      cnt = getCounterValue(state, subgroup);
      stateWeight += 1 / (cnt^2);
   }
}
```

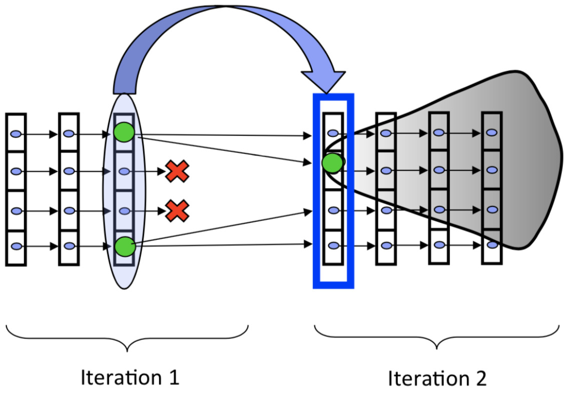
References: 1) “Rarity based guided state space search” – Ganai et al.; 2) “Using speculation for Sequential Equivalence checking” –Brayton et al

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)



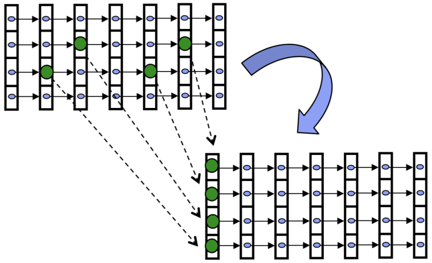
- Heuristic 1: Select Entire "Rarest Timestep"
  - Timestep-Based Rarity = Sum of rarity weights of each bit-parallel state in that timestep
  - Seed entire "rare bit-parallel vector" into next sim iteration
    - Like "backtrack to earlier timestep" and restart sim with new random inputs
  - BMC seeding: select rarest state in that rare timestep

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)



- Heuristic 2: Select Rarest States within "Rarest Timestep"
  - Select rarest timestep; sort states therein w.r.t. rarity
  - Selects top "n" rare states therein for next round of bit-parallel simulation
    - Recall: random input generation → higher coverage even when reusing same starting state
    - Bit-parallel sim evaluates them efficiently in parallel
  - BMC seeding: select the rarest state therein

Rarity-guided Simulation (Nalla et.al, ICCAD 2016)



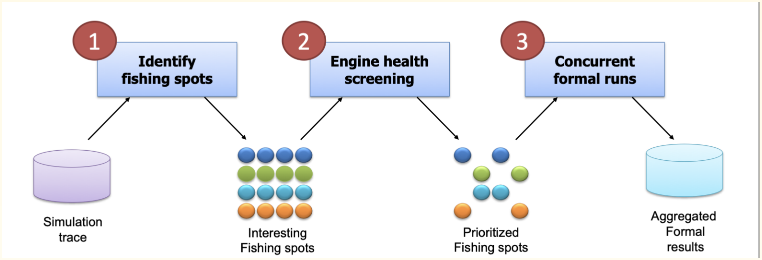
- Selection of best rare states across all timesteps
    - Particularly useful in constraint-based testbench: few vectors may be valid in a timestep
- Improves BMC coverage, since doesn't miss single-rarest state when selecting rarest timestep

Formal Bug Hunting with “River Fishing”

- Identify states in the design from where you can fish for bugs
- What is the selection criterion for *fishing spots*?
    - Outside interactions – e.g., standard protocol interfaces
    - Control and interrupts (FSMs, bus controllers, memory controllers)
    - Concurrent events (Arbiters, interrupts, schedulers, switches)
    - Feedback, loops, and counts (FIFOs, timers, counters, data transfers)

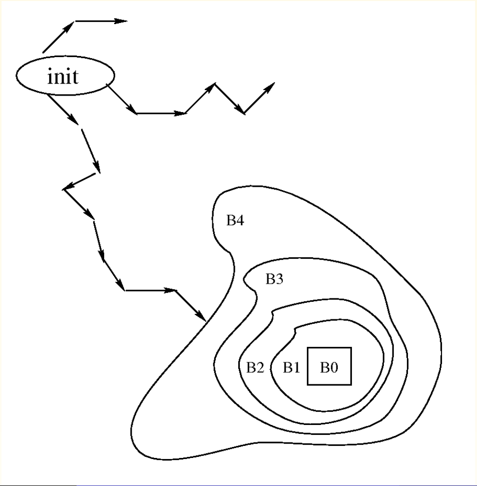
Formal Bug Hunting with "River Fishing"

- There might be a lot of fishing spots, how do you prioritize them?



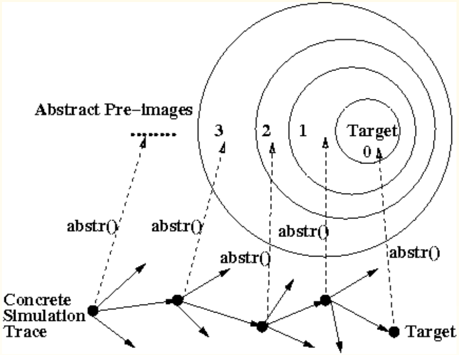
SFV by Augmenting Simulation: Other Approaches

- Target Enlargement: compute a few preimages of the target states, in order to create a larger set of target states
  - Improves the probability of hitting the target states
  - Simulation can use greedy search strategies



SFV by Guiding Simulation

1. Create an abstract design
2. Perform exhaustive formal search
   - Partition the reachable state set into onion rings
   - *i*-th onion ring can reach the target in *i* steps
3. Guide simulation to move the (concrete) simulation state to one that maps into the next closer onion ring
   - All concrete paths have corresponding abstract paths, but not vice-versa



SFV by Guiding Simulation: Challenges

1. The abstract design is too over-approximate
   - No legitimate concrete trace exists that maps to abstract trace
   - A short abstract trace may correspond to very lengthy concrete trace
2. A majority of concrete states hit are *dead-end* states
   - There is no path from *dead-end* state to a concrete state that maps to the next abstract onion ring

SFV by Guiding Simulation: Overcoming Challenges – Abstraction

- Abstraction is the key to overcoming the challenges
  - Retain enough behavior to efficiently guide simulation
  - Needs to be small enough to enable exhaustive search
- Use automatic abstraction techniques such as localization
  - CEGAR (counter-example guided abstraction refinement) can be used to automatically refine the abstraction
  - Applicable across wide-variety of testbenches
  - K. Nanshi and F. Somenzi. "Guiding simulation with increasingly refined abstract traces" DAC 2006.
  - Abstract models generated through localization tend to get large quickly
  - Exhaustive search quickly hits a brick wall

SFV by Guiding Simulation: Overcoming Challenges – Abstraction

- Abstract using domain knowledge
  - Verification engineer manually abstracts the design
  - Restricts the applicability of the technique
  - S. Shyam and V. Bertacco. "Distance-guided hybrid verification with GUIDO." DATE, 2006
  - F. M. de Paula and A. J. Hu. "EverLost: A flexible platform for industrial-strength abstraction-guided simulation" CAV'06
- Abstract using data mining and domain knowledge
  - Aims to avoid the pitfalls of manual approach and localization
  - A. Parikh, W. Wu and M. S. Hsiao, "Mining-Guided State Justification with Partitioned Navigation Tracks", ITC 2007
  - Applicability not demonstrated on industrial testbenches

SFV by Guiding Simulation: Overcoming Challenges – Abstraction

- Abstract away the data-path to retain control-path registers
  - Most designs have data paths and controllers
  - Most bugs result of infrequent interactions between controllers
  - Simulation attempts to explore as much of the control state space as possible, thus increasing the likelihood of finding bugs
  - Requires high-level design information
  - R. Sumners, J. Bhadra and J. A. Abraham, "Automatic Validation Test Generation using Extracted Control Models", VLSI Design, 2000

Industrial SFV: Methodology and Tool Support

- Synergistic Simulation/SFV Methodology needed
  - Common Design Model for Sim and SFV
    - There should not be any semantic gaps
  - Common Design Partitions / Units
    - FV typically applied to macro (sub-unit) level that require specifications at non-documented, fluid interfaces
  - Common Designer Assertion/Coverage Specifications
    - This has largely been achieved on the assertion side, how about coverage?
  - Common environment specification / Testbench drivers
    - Implies synthesizable testbenches?
- A tool that can scale to design partitions that have documented specifications

Industrial SFV: Formal Coverage

- Are We There Yet?
- What are the questions you want answered using coverage metrics?
    1. Does my FV environment allow all possible legal stimulus?
    2. Are the set of *Assertions* complete? Do they cover all possible design behavior?
    3. How much of the design space is covered by my proven assertions?
    4. Are the bounds obtained on undetermined assertions sufficient?

Industrial SFV: What are the Coverage events?

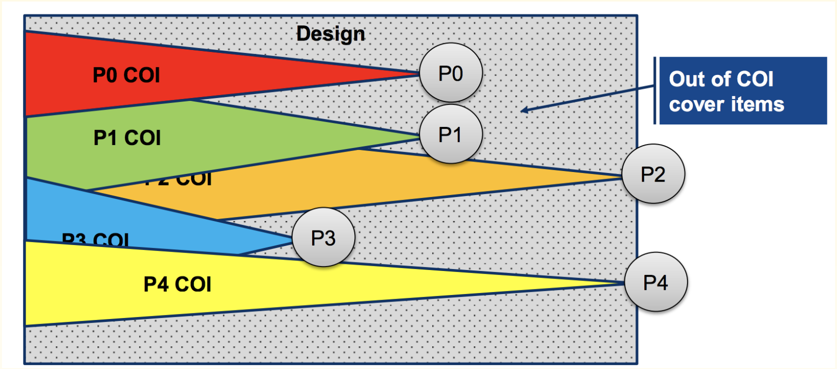
- Coverage events fall into two main buckets
  1. Code Coverage
     - Branch coverage
     - Statement coverage
     - Expression coverage
     - Toggle coverage
  2. Functional coverage
     - Property (SVA/PSL)
     - Covergroup

Industrial SFV: Stimulus coverage

- Attempt to hit all cover events
- Try to prove events that are unreachable
- Ideally all cover events must be hittable
- Unreachable cover events can be either due to:
  - Dead code: impossible for any stimulus to hit
  - Unreachable due to overconstraining environment
- Which engine is critical for ensuring stimulus coverage closure?
  - *Semi-Formal Engine ofcourse!*

Industrial SFV: Static Assertion COI coverage

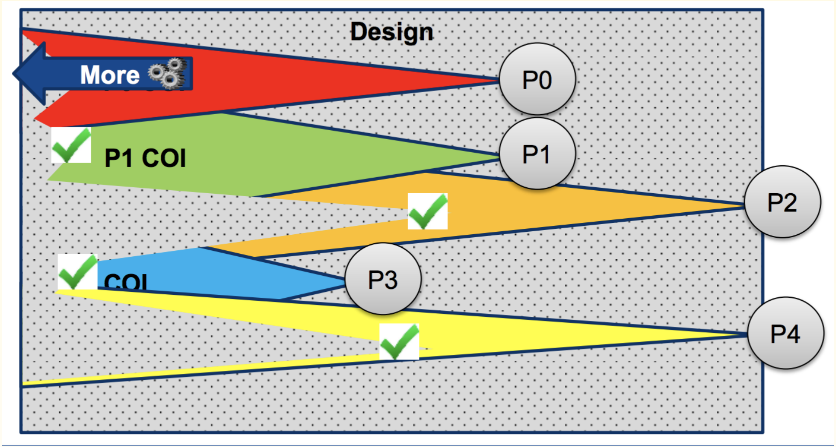
- If RTL has dead code, exclude the dead code from the analysis



Acknowledgment: Jasper Gold User Group 2018

Industrial SFV: Proof coverage

- SAT-based engines can generate a proof-core; logic responsible for proving correctness of a property

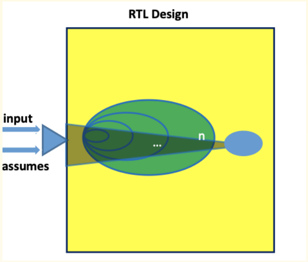


Acknowledgment: Jasper Gold User Group 2018

Industrial SFV: Are my bounds sufficient?

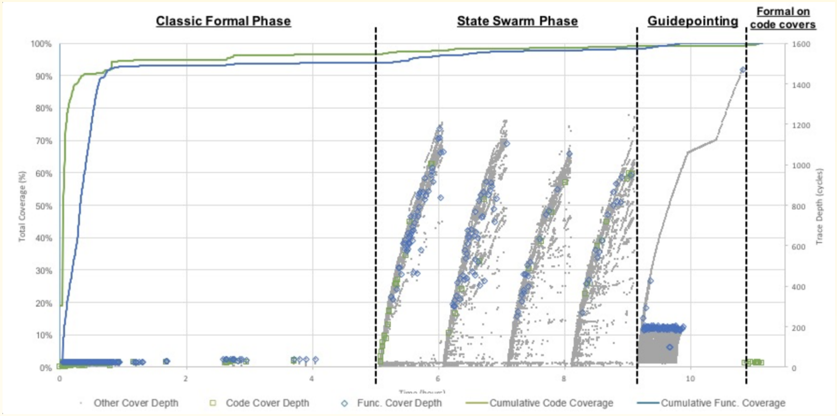
- “N” cycle bounded proof implies that all states reachable within “N” cycles have been analyzed
- Determining whether the bounds are sufficient is more Art than Science
- The current best approach is specifying functional cover events that force deep exploration of state space
    - Use SFV engines to hit the cover events
- Using a combination of designer knowledge and data from previous verification efforts, one can possible predict a good enough “N”
    - Possible application of ML/DL?

Industrial SFV: Are my bounds sufficient?



- Bounded proof core can be compared against structural COI
- Source code review to analyze the logic covered by the bound

Verification sign-off using Formal Verification



Acknowledgment: Jasper Gold User Group 2018

RuleBase-SixthSense: IBM's SFV toolset

https://www.research.ibm.com/haifa/projects/verification/SixthSense/

RuleBase-SixthSense is a system of cooperating algorithms

- Semi-Formal engines
- Formal engines
- Transformation engines: simplification / abstraction algorithms

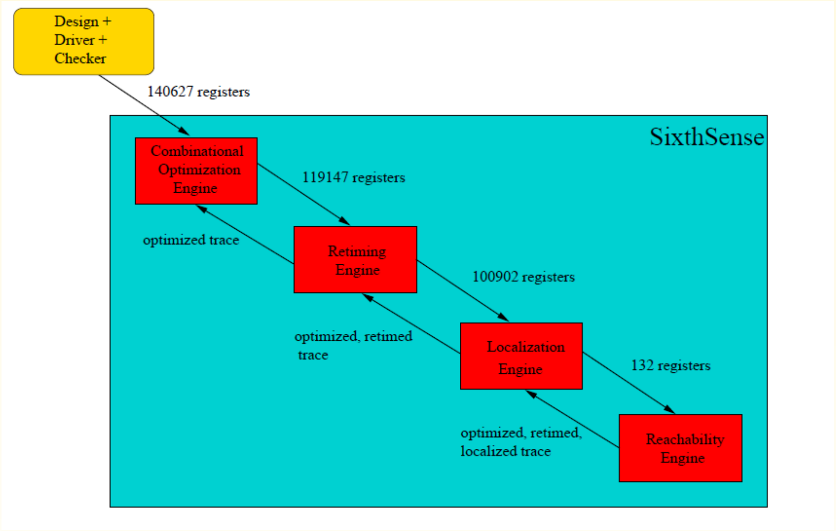
Transformation-Based Verification (TBV) framework

- Exploits maximal synergy between various algorithms
- Redundancy removal, retiming, induction, localization, ...
- Incrementally chop problem into simpler sub-problems until solvable

The Case for Transformation-based Verification

- FV – exhaustive, but needs exponential resources w.r.t size
- High performance design particularly difficult to verify
    - Speed, area and power concerns demand subtle optimization
    - Complex control, pipelining logic increases verification complexity
- Key Insight: Use automatic transformations to extract the simple underlying model
- RuleBase-SixthSense framework: Synergistically leverage various transformations to simplify and decompose complex problems

Example Transformation flow



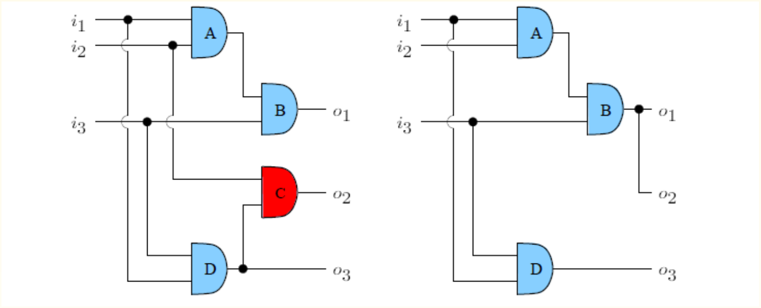
Example RuleBase-SixthSense Engines

- Boolean Reduction
- Sequential redundancy removal
- Min-area retiming
- Sequential rewriting
- Input reparameterization
- Localization
- Target enlargement
- State-transition folding
- Isomorphic property decomposition
- Fast Forward Engine
- Unfolding
- Liveness-to-Safety Transform
- Semi-formal search
- Symbolic simulation: SAT+BDDs
- Symbolic reachability
- Induction
- Interpolation
- Property Directed Invariant Generation (IC3)

Expert System Engine automates optimal engine sequence experimentation

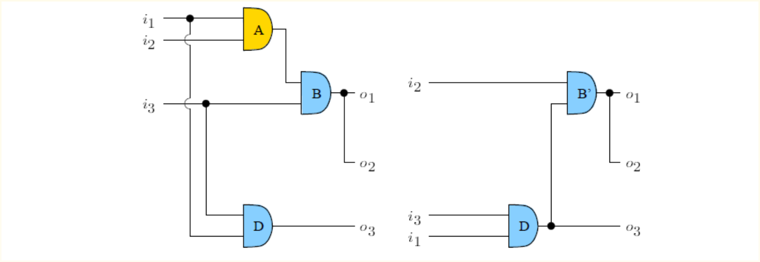
Boolean Reduction Engine (BRN)

- Combinational redundancy removal techniques: BDD- and SAT-sweeping
- Can be enhanced through leveraging observability don't cares
- Kuehlmann et. al., "Robust Boolean Reasoning For Equivalence Checking and Functional Property Verification", TCAD 2002.
- Kuehlmann et. al., " SAT Sweeping with Local Observability Don't-Cares", DAC 2006.



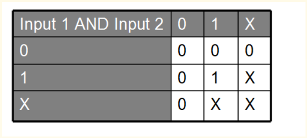
Boolean Reduction Engine (BRN)

- Logic rewriting algorithms, to simplify logic expressions
- Lowering gate count greatly enhances SAT-based reasoning
- Also tends to enhance reduction potential of other algorithms
- A. Mishchenko et. al., “DAG-aware AIG rewriting a fresh look at combinational logic synthesis”, DAC 2006

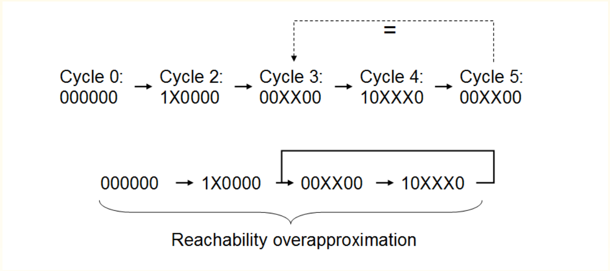


Boolean Reduction Engine (BRN)

- Ternary Simulation: simulate an AIG over 3-valued logic
  - Sequence of 3-valued states
  - Converge when the current state is a subset of past states
- Over-approximate reachable state set can be used for identifying constants/equivalent signals



Input 1 AND Input 2	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

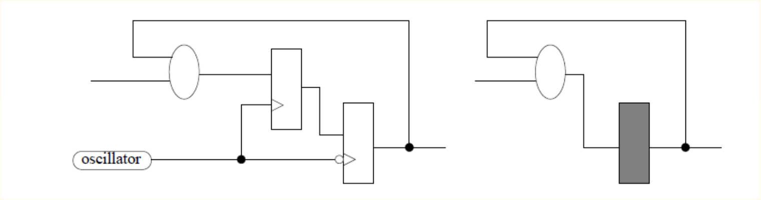


Sequential Redundancy Removal (EQV)

- Sequential Redundancy Removal Algorithm: A Recap
  1. Guess a set of redundancy candidates; sets of gates that are expected to be functionally equivalent (modulo inversion)
  2. Attempt to prove redundancy candidates accurate
  3. If any redundancy candidate cannot be proven, partition the groups to separate those that cannot be proven equivalent; goto Step 2
  4. The current groups reflect true redundancy; simplify the netlist
- Superset of BRN reductions, more expensive than BRN
- Very effective for bug-finding as well as proofs
- Mony et. al., "Exploiting Suspected Redundancy without Proving It", DAC 2005
- Mony et. al., "Speculative-Reduction based Scalable Redundancy Identification", DATE 2009

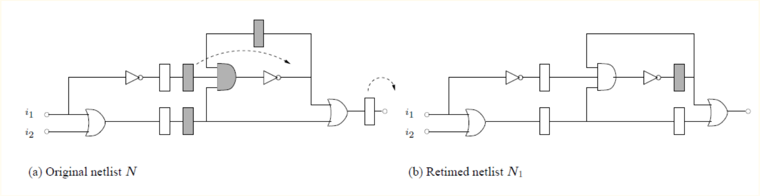
State Transition Folding (MOD)

- MOD: a structural state folding engine
  - Each state transition in the transformed design corresponds to multiple original transitions
  - Used to perform clock abstraction
    - Replaces *Master-Slave* with a *flip-flop*
  - Very powerful in designs with multiple clock domains
  - Reduces the minimum depth at which targets can be hit
  - Per Bjesse, James H. Kukula: "Automatic generalized phase abstraction for formal verification", ICCAD 2005



Transformation (RETiming)

- RET: Min-area retiming: Reduces the number of registers by moving them across combinational gates
  - Very powerful for deeply-pipelined as well as feed forward designs
  - May increase AND and Input count, use in association with BRN, CUT
  - J. Baumgartner and A. Kuehlmann, "Min-Area Retiming on Flexible Circuit Structures", ICCAD 2001
  - A. Kuehlmann and J. Baumgartner, "Transformation-Based Verification Using Generalized Retiming", CAV 2001



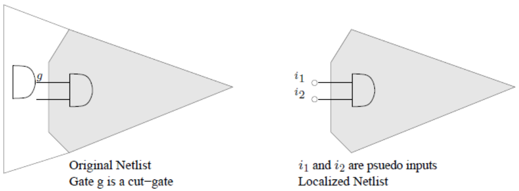
(a) Original netlist N (b) Retimed netlist N_1

Liveness to Safety (LIV)

- LIV: Liveness to Safety transformation
    - Properties categorized as either *Safety* or *Liveness*
    - *Safety*: Is my division result correct?
        - Finite counterexample
    - *Liveness*: Will the request eventually get a grant?
        - Infinite counterexample: to illustrate that grant can never occur
    - Traditional liveness checking very expensive
    - Can transform the netlist to convert liveness to a safety property
        - But doubles the number of state elements in the design
- Armin Biere, Cyrille Artho, Viktor Schuppan: "Liveness Checking as Safety Checking"

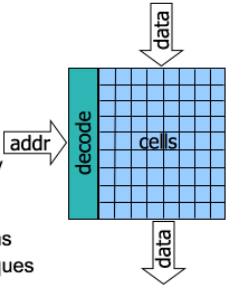
Transformation (Localization)

- AXE: localization; remove logic not relevant to the target
  - Overapproximate – proofs in localized netlist valid
  - Necessary to complete proofs on huge designs
- Core Idea: Use SATisfiability-based analysis to identify logic needed to prove target unreachable for specific number of cycles (say N)



Native Memory Modeling

- Native memory array support
  - *No more bit-blasting!*
  - **Much** faster semi-formal bug hunting
  - Proof complexity ≈ #reads relevant to property
    - Not #addresses in array
  - Enables reasoning about software-like systems
  - Enables various advanced abstraction techniques



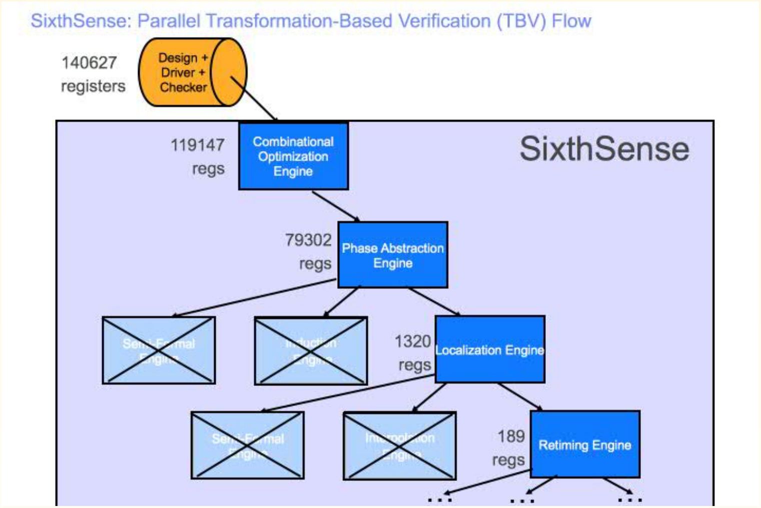
Expert System (XPT)

- Discovering well-tuned transformation flow non-trivial
  - Need to understand engines/options, rate their effectiveness
  - Intelligent experimentation necessary to find conclusive flow
- Don't want to delve into the engines $\Rightarrow$ use the XPT engine
- XPT automates the experimentation needed to solve tough problems
- Every problem is different, can the XPT engine itself be tuned?
  - **YES!**
- Mony et. al., "Scalable Automated Verification via Expert-System Guided Transformations", FMCAD 2004.

Why Parallel XPT

- Discovering a well-tuned engine flow is nontrivial
- Every problem different; complex problem requires 100's of transforms
- XPT automates this experimentation by deploying rules
    - Deployed rules setup for commonly encountered problems
    - Deployed rules not aggressive
    - Certain strategies only work for a small subset of problems
    - XPT tries multiple strategies – but in prioritized manner
- Solution: Parallel XPT with multiple strategies in coordinated manner

Parallel Orchestration



SFV Algorithms in RuleBase-SixthSense

Augmenting Simulation

- Ability to manually define guideposts and have the tool step through them
- Target Enlargement
- Interleaved BMC and random simulation
    - State prioritization
    - Automatic generation of lighthouses

Guiding Simulation

- Automatic generation and refinement of abstract models using localization
- Ability to tunnel between onion rings using BMC
- Heuristic guidance strategy to avoid deadend states

Applications of SFV

Virtually all RuleBase-SixthSense applications benefit from semi-formal search

Assertion-based verification

- Typically done by designers with lesser experience level with FV and toolset
- Testbenches developed with little thought about "proof strategy"
- SFV very useful to wring out bugs

Reference-model based verification

- Comprehensive checks, usually implemented as an abstract reference model
- Critical for verification sign-off; find corner case bugs

Applications, Cont'd

Silicon-failure recreation efforts: When a chip misbehaves . . .

- On-chip debug facilities offer partial insight into cause
- Usually have a good idea of property to check, "buggy region"
- SFV very useful since often requires a fairly large design slice, and bug-hunting vs. proving is "the mission"

Coverage analysis

- Leverage formal algorithms to help simulation reach hard-to-hit scenarios

Sequential equivalence checking

- Semi-formal search useful to find mismatches, assist in guessing equivalent gates

SFV: Conclusions and Future

- SFV is an enabling technology for wide-spread FV usage
- FV-based Verification sign-off is impossible without SFV

Future Research

- The future of SFV is bright!
- Take advantage of new machine architectures for improving simulation throughput (bit-parallel simulation)
- Enhance simulation through intelligent pattern generation
- Methodology/Algorithms to determine if bounded coverage is sufficient