Assertion-Based Formal Property Verification: Making it Practical in Real Life

Erik Seligman
Intel Corporation
DAC Workshop: “Beyond Syntax And Semantics”
June 9, 2008
Agenda

- Introduction
- Principle #1: Highly Usable Libraries
- Principle #2: Make Assertions Happen
- Principle #3: Lay Groundwork For Formal
- Principle #4: Make Formal Verification Work
- Results & Conclusions
Agenda

• Introduction
• Principle #1: Highly Usable Libraries
• Principle #2: Make Assertions Happen
• Principle #3: Lay Groundwork For Formal
• Principle #4: Make Formal Verification Work
• Results & Conclusions
Motivation

• Validation crisis
  – ➔ Need quantum leap in productivity

• Assertions are well-accepted
  – “Executable comments”
  – Help find & locate bugs early

• Formal Verification (FV): source of the leap?
  – Mathematically prove, not just simulate
  – Theoretically complete coverage
    – Practical limitations, but complete in subspace
  – Available early in projects
    – No dependence on test env
  – If done right, should reduce need for tests
Pitfalls

• Naïve attempts to introduce FV often fail
  – Need infrastructure for property creation
  – Designers need motivation for properties
  – Thought processes different for formal
  – Teams tend to revert to known methods

• Need to think through these issues

• Plan infrastructure & processes to enable
  – Assumption: You already know about assertions
  – Challenge is to effectively combine with FV
Agenda

- Introduction
- Principle #1: Highly Usable Libraries
- Principle #2: Make Assertions Happen
- Principle #3: Lay Groundwork For Formal
- Principle #4: Make Formal Verification Work
- Results & Conclusions
Need For Assertion Libraries

• Assertion languages often too general
  – Designed by PhDs to cover all cases
  – Provide building blocks, not designer tools
  – Average designer has hands full

• Examples of SVA gotchas
  – Glitch dangers
    always @(a or b) assert (a != b);
  – Sequences vs triggering
    assert property (a ##1 b);
    assert property (a |=> b);
  – Operator definitions
    assert property (s1 and s2);
    assert property (s1 intersect s2);
Popular Library Solution: OVL

assert_always
#(`OVL_ERROR, `OVL_ASSERT, "Bogus grant")
al(clk, reset_n, req||!grant);

• Each assert is a cleanly defined module

• Good, but still room for improvement
  – Modules can’t be used in SV procedures
  – All arguments need to be typed
  – Need to explicitly specify clk/rst
  – Combo asserts can still glitch if misused
always_ff @(posedge clk) begin
    ...
    if (foo) begin
        ...
        if (bar) begin
            ...
            v1 <= messy_function(b,c,d,e,f);
            // want assertion on v1 here
            ...
        end
        ...
    end
    ...
end
// OVL assertion on v1 has to go here!
Improved SVA Libraries

```
`ASSERT_VERIFY(req|!gnt, clk, rst)
`ERR_MSG("Bogus grant");
```

- Macro version: improvements over OVL
  - Macro can be placed in procedural code
  - Arguments can be any type (even property)
- Most Intel projects moving to macro library

- WIP in IEEE p1800 WG for SVA extensions
  - Checkers, glitch-protection, inferred clks, LTL, etc.
  - Will enable fully parsed library with same benefits
Agenda

- Introduction
- Principle #1: Highly Usable Libraries
- Principle #2: Make Assertions Happen
- Principle #3: Lay Groundwork For Formal
- Principle #4: Make Formal Verification Work
- Results & Conclusions
Assertions: Part of the Design Process

- Define standard assertion note
  
  // Assertion a123: Check for legal grants;

- Designer adds: spec, testplans, RTL
  - Assertion idea != interrupt thought flow

- Assertion expert role
  - Scripts to collect assertion notes
  - Help designer implement/focus

- Assertions: casual & easy
  - Pitfall: Avoid requirements seen as penalty
    - “Must eventually prove X% formally”
“Free” Assertions

• Reuse blocks ➔ standard assertions
  – Intel chipset group: form for std FSM
  – Tool creates RTL template + assertions
    – FSM never stuck
    – All states reachable

• Assertion packages with IP building blocks
  – AMBA, PCI asserts from vendors?
  – Develop asserts for internal standards
Encouraging Useful Assertions

• Focus on high-level intent
  – Assertions = “executable comments”
  – Add insight to design
    – Micro-assert on a couple of RTL lines less useful

• Don’t be afraid of some modeling code
  – Auxiliary calculations / wires are fine
    – Provide `ifdef to exclude from synthesis
  – Full reference models in areas of concern
  – Smaller “shadow models” often very useful
"Shadow Model" Example (PCIE)

- Not full ref model, but enables good asserts
  - *End of transaction* \(\Rightarrow\) *commit within n cycles*
  - FV found serious chipset bugs missed in sims!
Agenda

- Introduction
- Principle #1: Highly Usable Libraries
- Principle #2: Make Assertions Happen
- Principle #3: Lay Groundwork For Formal
- Principle #4: Make Formal Verification Work
- Results & Conclusions
Effective Formal Verification: Prerequisites

- Include FV in testplans
  - Part of validation, NOT out-of-band ‘extra’
  - Pick good FV battles:
    - Feasible size, Good assertions, Many corner cases

- FV by model expert
  - FV reveals obscure corner cases
  - Need deep understanding

- Cover points need to be in place
  - Describe typical “interesting” situations
  - Help judge FV env effectiveness
Preparing The FV Infrastructure

• FV is different for a simulation team
  – “Plug & Play?” – sort of, but watch details...

• Choose highly usable tools
  – Main FV debug activity: analyze counterexample
    – Counterexample (CEX) = waveform violating assertions
    – Fancy debug support: Wave views, what-if, multi-cex?
  – Same assertions in FV and simulation

• Project-specific glue
  – Wrapper scripts based on simulation env
  – Set up default clock/reset config
Agenda

- Introduction
- Principle #1: Highly Usable Libraries
- Principle #2: Make Assertions Happen
- Principle #3: Lay Groundwork For Formal
- Principle #4: Make Formal Verification Work
- Results & Conclusions
Starting FV On A Design

• Initial runs are a feasibility test
  – Complexity of FV hard to predict
    – Not proportional to transistor count
    – Small arithmetic can be very hard!
  – If time/memory explodes, reconsider target
    – Lower hierarchy?
    – Blackbox/prune parts of model?

• Also, look early at cover point proofs
  – FV tool finds scenario to hit each cover point?
  – Cover point not reachable ➔ overconstrained!
    – Important quality check
    – Repeat after new assumptions added
Assumption Creation Loop: Majority of FV Time

- Mindset change from sim; prepare team!
  - Early runs have many false negatives
- More assumptions => more interesting CEX
  - Interesting bugs not found on first run
  - Several rounds of assumes ➔ deep traces
  - Be sure to check assumptions too, in simulation or FV
Example: Adding an Assumption

- In one unit, many assertions failed
  - Mid-transaction address changes
- Needed input assumption to prove
  - Bug found when assumption fired in simulation!

RTL Under Test (several cycles per transaction)

ASSUME (held for $n$ cycles)
Assumption Count Exploding?–Don’t Give Up Yet

• Possible bad choice of boundary
  – Effectively reimplementing neighbor block?

• Consider increasing hierarchy level
  – Add upper level & many blackboxes?

• Also consider simplifying problem
  – Only cover certain modes
    – Example: PCIE: prove for x16, not x4, x8?
  – Restrict data
    – Will one nonzero bus bit test most major logic?
    – Are fully general packet payloads needed?
Example: FV too hard?

MPE = Memory Protocol Engine
MRA = Memory Read Arbiter
Correct Hierarchy Makes FV Easy

MPE0 — MRA0 — MRA1 — MPE1

MSB
Complexity Issues (Time, Memory)

• As mentioned before, hard to predict
  – Issue may appear after adding assumptions

• Various strategies available
  – Try running at lower hierarchy
  – Consider blackboxing or freeing logic
  – Restrict modes or data
  – Reduce size of parameterized elements

• Eventually may need to waive
  – Again, don’t give up too soon
  – FV of partial design space still useful!
Bounded vs Full Proofs

• Which of these do you need?
  – “Assertion can NEVER be violated”
  – “Assertion can never be violated by any possible simulation of length up to <n>”

• Bounded proof usually easier for tools
  – Use cover point proofs to judge good bound
  – Bound == lengths of interesting scenarios
  – Some coverage lost vs full proofs
    – But often at point of diminishing ROI

• Consider modifying starting state
  – Fill queue at start of proof...?
Agenda

• Introduction
• Principle #1: Highly Usable Libraries
• Principle #2: Make Assertions Happen
• Principle #3: Lay Groundwork For Formal
• Principle #4: Make Formal Verification Work
• Results & Conclusions
Results & Conclusions

• Assertion-based FV used effectively in Intel
  – Chipsets: Found numerous bugs missed in simulation (PCIe, memory controllers)
    – Also uncovered basic flaw in one validation env
  – CPU designs expanding this usage mode
    – Competitive: Recent project devoted 8% of validation resources in front-end design, found 8% of bugs
    – 30-35% of bugs found by assertion FV were unlikely to be found in simulation

• Future projects improving systematic use
  – Better integrating FV into validation plans
  – Improved assertion libraries for designers
Overall Recommendations

- Assertion-based FV: important part of design
- Plan for it from the beginning
- Major principles
  - #1: Highly Usable Libraries
  - #2: Make Assertions Happen
  - #3: Lay Groundwork For Formal
  - #4: Make Formal Verification Work
Acknowledgements

- Many slides inspired by work from or with fellow Intel engineers Mehbub Ali, Dhruba Kalita, Alon Flaischer, Dan Carmi, Adriana Wolffberg, Dmitry Korchemny, Rami Naqib, Kapilan Maheswaran, Ram Koganti
Backup Slides
What is OVL?

- Full info available at www.eda.org/ovl.
- Usage example:
  ```
  assert_never #(1,0,"problem seen")
  myassert(bclkmain,resetnn,foo!=bar);
  ```
- More assertion examples:
  - Simple boolean conditions: assert_never
  - Bit vector checking: assert_{one,zero_one}_{hot,cold}
  - Check behavior between events: assert_win_change
  - Sequential behavior: assert_next, assert_time,
  - Track signal value changes: assert_delta
  - More complex behaviors: assert_fifo_index
Definition of `ASSERT_VERIFY

property verify(prop, clk=`default_clk, rst=1'b0);
   @(clk) disable iff (rst) prop;
endproperty : verify

``define ASSERT_VERIFY(prop, clk, rst) \ 
   assert property(verify(prop, clk, rst))"
Further Library Improvements

- WIP on SVA standard extensions
  - IEEE p1800 2008 working group

- Need good container for library properties
  - Macros have many disadvantages
  - Defining “checker” – module-like, but OK in procedural code

- “Deferred Assertions”: Glitch-free combo asserts

- LTL added to SVA language (eventually, always, nexttime, …)