Verification-Guided Hierarchical Design

Robert Kurshan
NYU - Amir Pnueli Memorial Lectures
May 8, 2010
Simulation coverage does not scale gracefully with growing system design size. Component interactions grow exponentially with the number of system components, while conventional system test at best can increase coverage as a linear function of allotted test time.

Likewise, capacity limitations are commonly cited as the essential gating factor that restricts the application of model checking to at most a few design blocks.
Design Costs

- synthesis/layout < 50% total cost
  - more or less linear in chip size
- debug/verification = 50% - 80% of total cost
  - embedded software
  - n parallel components of size m leads to m^n system states
  - so functional verification grows exponentially with design size
- widely held that the cost of fixing a bug grows exponentially with the development stage at which it is detected.fixed
  - on account of increasing interactions with other components that also must reflect changes from fixes
- holding down costs leads to less test coverage and lower design reliability
  - but cost of product failures can also be high
  - ready for a $500M recall? (Intel FDIV bug)
  - How about $2B? (Toyota sudden acceleration – likely a s/w bug)
Abstraction has long been used successfully in pilot projects to apply model checking to entire systems. Abstraction in conjunction with guided-random simulation can be used in the same way to increase coverage for conventional test.
Abstraction as Divide-and-Conquer

- Divide-and-conquer requires the precision of formal methods

- Types of divide-and-conquer
  - Horizontal (flat) decomposition – abstracts component environment
  - Vertical (hierarchical) decomposition – abstracts lower-level details

- Conservative vertical abstractions support verify-only-once: at highest level of abstraction where property is defined
  - Contrast with Transaction-Level Modeling

- Enables earlier debug
  - main power and innovation will come from vertical decomposition
Vertical (Hierarchical) Decomposition

• Design development today: **data before control**
  – Controllers need to point to defined data structures
  – But: upside down – often need to modify data structures for controllers

• Decompose vertically: **control before data**
  – Use stubs as place-holders for data
  – Controllers point to stubs
  – Stubs are oracles for data path computation

• Imposes hierarchical decomposition
  – Control at higher levels (coarse granularity supports global verification)
  – Data paths at lower levels (fine granularity verified locally)
  – Constant complexity at each level – scales with increasing design size
Vertical (Hierarchical) Decomposition, cont.

1. Start with functional spec, floor plan, etc
2. Derive properties (test plan) BEFORE coding design!
   1. Formal spec with comments
   2. Specification reviews (like design reviews) for completeness
3. Partition properties into levels
   1. Control properties first (global properties)
   2. Data path properties last (local properties)
4. Code to properties
   1. Use stubs as place-holders/oracles for lower levels
   2. Verify (simulation or formal) as you design

Implements top/down – bottom/up hierarchical design process
20 years ago …

Packet Layer Controller chip development at Bell Labs

- 200,000 transistors
- Developed entirely under the control of formal verification through a top/down stepwise refinement hierarchy
- 20% of projected cost
  6 staff years/2 calendar years vs projected 30 staff years
- “reliability of a 2nd generic release”
Abstraction is **more general** than subroutine

- e.g., abstraction may never terminate
**Refinement Step**

- Use non-deterministic delay as place-holder for to-be-defined procedure
- Use non-deterministic branch to model possible returns from abstract procedure

Conservative abstraction of refinement: verify property only once, at highest level it’s defined

**Stub types**
- Data
- Datapath
- Control

datapath = datastructure
control = FSM
Example: stubbing a FIFO – Lv1

Lv1 data abstraction: track msg1, All others -> msg0
Lv1 Assertion: After (APPL.put_msg1)
Eventually(msg1_on_bus)
[verifies MGR]

Lv1 Constraint: After (FIFO.tail=msg1)
Assume Eventually (FIFO.head=msg1)

thanks Chris Komar
Example: stubbing a FIFO – Lv2

Lv2 refines FIFO stub into 2 sub-stubs
single msg1 can enter either
--------------------------------------------------

Lv2 Assertion: After \((FIFO.tail=msg1)\)
Eventually \((FIFO.head=msg1)\)
\((=\text{Lv1 constraint})\)
[Checks that FIFO MGR prevents starvation]

Lv2 Constraints:
After \((FIFO1.tail=msg1)\) Assume Eventually 
\((FIFO1.head-1=msg1)\)
After \((FIFO2.tail=msg1)\) Assume Eventually 
\((FIFO2.head=msg1)\)

thanks Vic Du
Further Refinements

Lv3: add FIFO mechanism (head/tail pointers)
   - verify succession for real stages + abstract stage
     abstracting any number of words
     (verifies Lv2 constraints)

Lv4: expand abstract stage to full length of FIFO
   - succession property follows inductively

Lv5: expand stages to full word width
   - succession property follows inductively
Consequences

- Design and verification done together
  \[\rightarrow\] earlier hence cheaper debug
  -- D sees bugs as they’re encoded (not months later)
  -- debug when design is simpler, hence easier to fix (fewer adjacent consequences)
- PV promoted to S/VE
- D designs global flow control before low-level data structures (iteratively)
  \[\rightarrow\] Designer focuses on function before structure
  -- structure serves function (today it’s reverse)
    eg, requirements for memory coherence will precede and define requirements for a cache protocol (rather than reverse)
- Coverage/Capacity scales linearly with design size
Hierarchical Flow

Designer (D) ←→ Specification/Verification Engineer (S/VE) work hand-in-hand


3. D writes flow-control code to support testing of Lv1 assertions, writes stubs for associated lower-level (Lv2) structures

4. S/VE writes constraints on stubs (= Lv2* assertions)

4. D runs quick checks (limited/automatic model checking) of Lv1.
   -- debugs/fixes/iterates 4.
   -- S/VE works on DNFs: mc + reduction/decomp, hybrid, guided-random

Iterate 2, 3, 4.

5. D refines Lv1 stubs into Lv2 code to support testing of Lv2 assertions + Lv2 stubs (ie, 5. = 2. one level down)