

Model *Sim*®

**Assertion Based
Verification with PSL**

**Mentor
Graphics**®

ModelSim PSL Agenda



■ PSL

- What is PSL?
- PSL Language Basics
- Property-assisted design & verification methodology
- Assertions and ModelSim
- Functional Coverage and ModelSim
- PSL Code Examples
- PSL Demo
- PSL Compared to Other Solutions (optional if time)

What is PSL?



- **PSL: Property Specification Language**
 - **Derived from IBM’s Sugar language**
 - Donated to Accellera as a basis for PSL
 - **Sugar preferred over other donations:**
 - ForSpec from Intel
 - CBV from Motorola
 - Others
 - **Do not confuse PSL and Sugar names**
 - IBM indicates acceptance of PSL as the latest Sugar version
 - But still calls it Sugar
 - No way to externally verify what IBM is doing internally
 - At least one EDA vendor refers to “Sugar PSL” support
 - Actually means support of a mixed Sugar (pre-PSL) and PSL syntax
 - Bottom line: We use “PSL” when referring to the Accellera standard.

What Differentiates PSL?



- It's an Accellera standard (version 1.1)
- It works with Verilog, VHDL, and System C
- Easy to learn, read, and write
- Mathematically rigorous and expressive
- Easy to translate from natural language specs
- A single language which encompasses multiple verification methodologies
 - Assertion based
 - Functional coverage
 - Formal

Some Definitions



- **Boolean**
 - Expression evaluating True or False
 - Basic building block of sequences
- **Sequences**
 - Specification of how boolean expressions relate to each other
 - Relationship can be over time (temporal)
- **Property**
 - Functional specification of behavior
 - Can describe good (always) or bad (never) behavior
- **Assertion**
 - Directive that a property must always or never hold
 - Mainly use to detect undesired (bad) design behavior
- **Cover**
 - Directive that occurrences of a sequence should be counted
 - Enables Functional Coverage metrics
 - Used to indicate that desired/required design behavior actually occurs during simulation

Definitions (2)



- **Assertion-based verification**
 - A verification methodology that ensures functional specifications (properties) of a design are not violated
 - Formal tools can prove a property will hold
 - Simulators report if a property does not hold

- **Property-assisted verification**
 - Assertions are one way to exploit properties
 - Other opportunities for exploitation:
 - Functional coverage
 - Stimulus generation

What's The Problem?

■ The Verification Gap

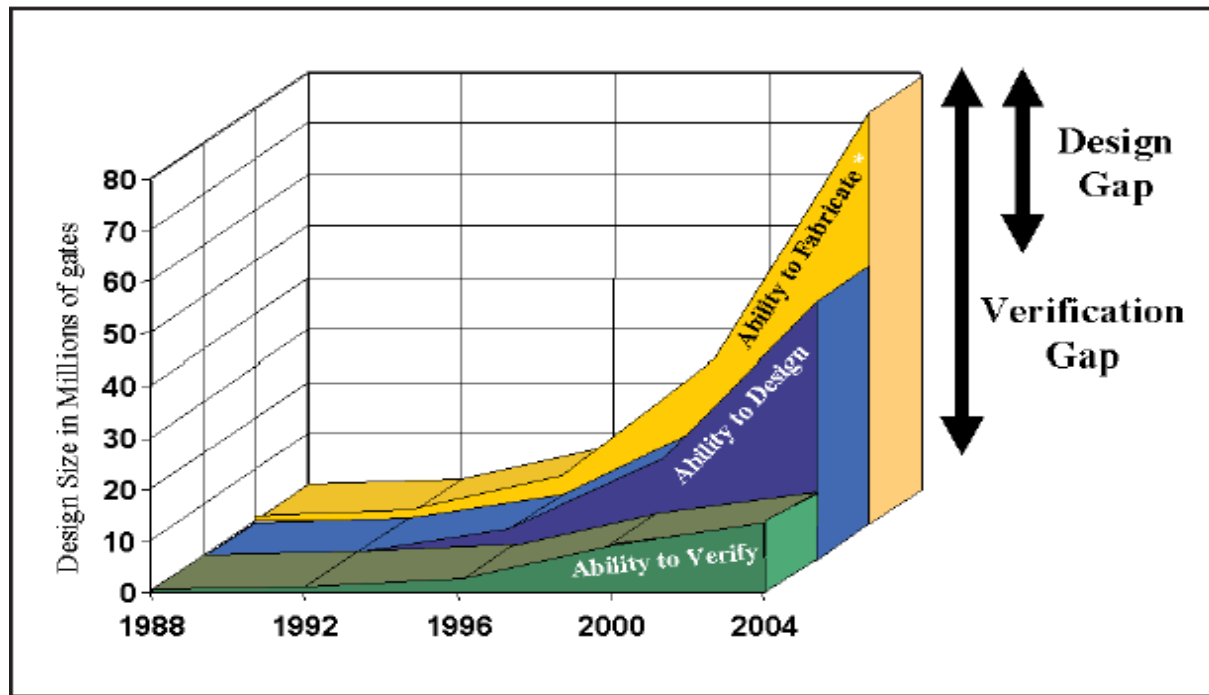


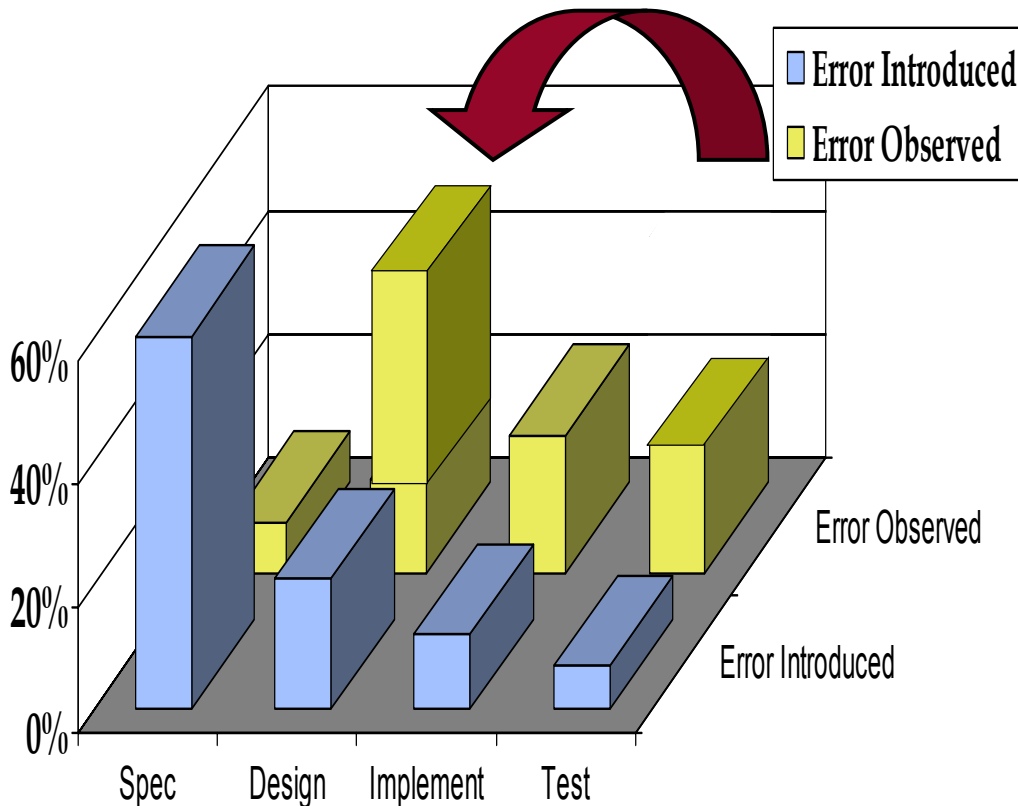
Figure 1. The verification gap leaves design potential unrealized.
(* source: SLA Roadmap, 2001)

Why the Gap Exists



- **Moore's Law**
 - EDA vendors driven by silicon/SoC methodologies vs. verification
- **Lack of Standards**
 - No portability between projects or companies
- **Verification productivity**
 - Exhaustive simulation impossible
 - What stimulus to generate and how to control it?
 - Block and system verification disconnects
 - Different testbenches causes inability to easily reproduce errors
 - Creating testbenches & tests as challenging as chip design
- **Verification Completeness**
 - Traditional source and toggle coverage != functional coverage
 - IP Blocks typically treated as black boxes
 - Were deep states or corner cases covered?
- **Debug Productivity**
 - Did error propagate to output?
 - How long after cause was error seen on output
 - Where did the error occur?

PSL Bridges the Gap



- **1st standard property specification language**
 - Formal specifications (Mil-Aero “shall”)
 - Executable specifications?
 - Static & Dynamic verification
- **Increases verification productivity**
 - Documents requirements and assumptions
 - Guide & direct stimulus generation
 - System and block verification
- **Measuring Verification Completeness**
 - Functional coverage not code coverage
 - Check corner cases and deep states
 - Allows controlled white-box verification of IP
- **Increases Debug Productivity**
 - White box: Bugs don't have to propagate to outputs
 - Bugs identified earlier
 - And closer to the source of the problem

Assertions Really Do Work



Assertion Monitors	34%
Cache Coherency Checkers	9%
Register File Trace Compares	8%
Memory State Compare	7%
End-of-Run State Compare	6%
PC Trace Compare	4%
Self-Checking Test	11%
Simulation Output Inspection	7%
Simulation Hang	6%
Other	8%

Kantrowitz and Noack [DAC 1996]

Assertion Monitors	25%
Register Mismatches	22%
Simulation "No Progress"	15%
PC Mismatch	14%
Memory State Mismatch	8%
Manual Inspection	6%
Self Checking Test	5%
Cache Coherency Check	3%
SAVES Check	2%

Taylor et al. [DAC 1998]

Assertions in real designs:

34% of all bugs found were identified by assertions on DEC Alpha 21164 project.

[Kantrowitz and Noack DAC 1996]

17% of all bugs found were identified by assertions on Cyrix M3(p1) Project

[Kronik '98]

25% of all bugs found were identified by assertions on Dec Alpha 21264 project

[Taylor et al. DAC 1998]

50% of all bugs were identified by assertions on Cyrix M3(p2) Project

[Kronik '98]

85% of all bugs were found using over 4000 assertions on HP Project

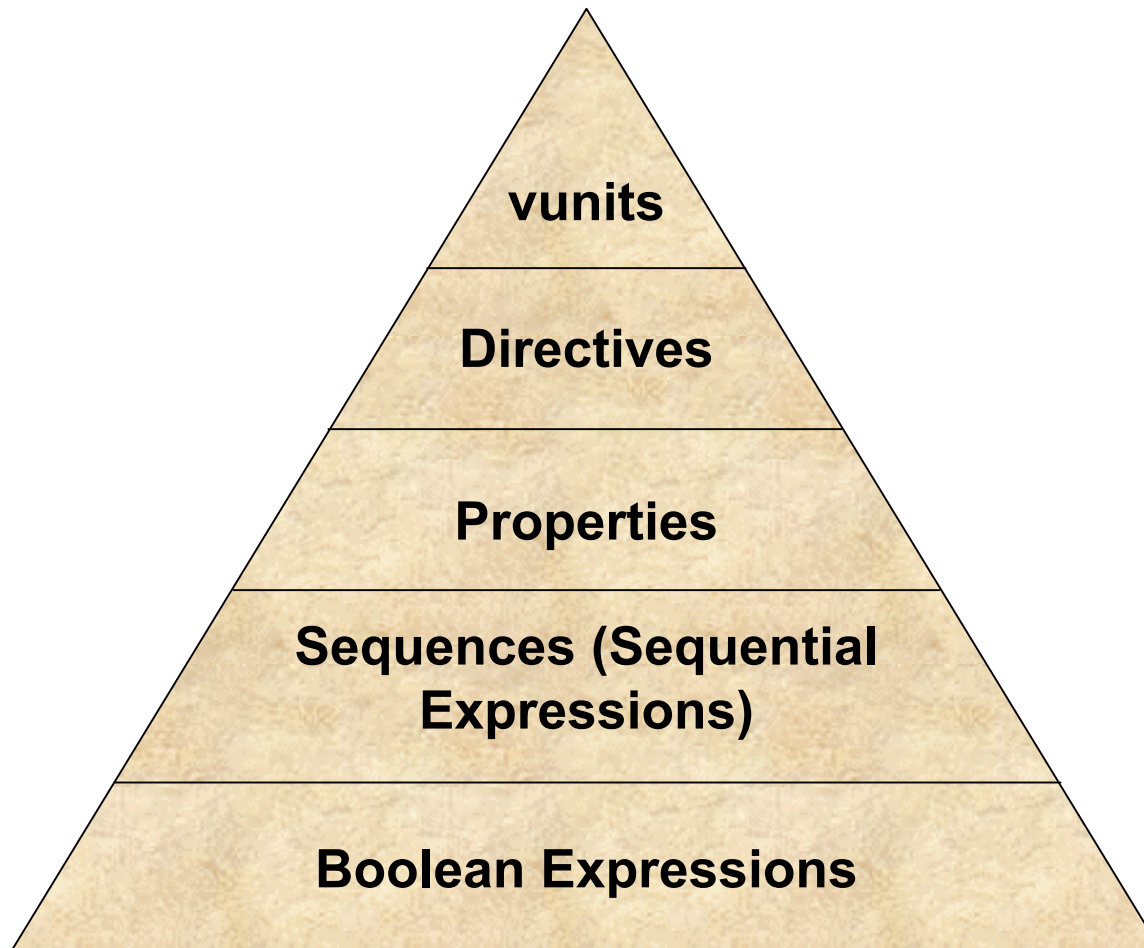
[Foster and Coelno HDLCon 2000]

10,000 assertions in Cisco project

[Sean Smith 2002]

Percent of design bugs found via assertions is growing!!

PSL Language Structure



PSL Has Flavors



- **PSL was designed to work with any other Hardware Description Language**
- **Each language that works with PSL has its own “flavor”**
 - Verilog
 - VHDL
 - GDL (General Design Language)
 - How SystemC support will be provided
- **What’s important to know about the flavors?**
 - **Boolean expressions are written and have the same semantic meaning as the language flavor**
 - Any expression legal as the condition in the language’s if-statement is a boolean expression
 - **Some syntactic differences**
 - “is” versus “=”
 - **Everything else is PSL-specific and HD(V)L independent**

Boolean Expressions



■ What is a Boolean

- Any expression legal as the condition of an if-statement in the language
- Evaluate true or false
 - Verilog - True (1'b1) or False (1'b0)
 - `!(a & B)`
 - VHDL - Boolean'(FALSE or TRUE), BIT>('0' or '1') and STD_ULOGIC('0' or '1')
 - `not(a and b)`
- No temporal (time relationship) associated with boolean expression
- Boolean implication:
 - `a -> b` if a then same cycle b
 - `a -> next b` if a then next cycle b
 - `a <-> b` if and only if, a implies b and b implies a

Sequences



■ Sequences are boolean expressions related over time

– Delimited by curly braces {...}

```
{a; b; c} -- a, one cycle later b, one cycle later c
{a; [*0 to 2]; b; [*]; c} -- a, then b in next cycle or up to 3 cycles
-- later, c sometime after b (eventually)
{a; b[->]; c} -- a, then sometime later b, then c
-- in the next cycle
```

– Sequential implication (note: | differentiates sequential from boolean implication!)

```
{a; b} |-> {c; d} -- RHS starts eval in last cycle of LHS is true
{a; b} |=> {c; d} -- RHS starts eval in cycle after LHS is true
```

If LHS is a sequence, must use sequential implication

■ Clocked sequence:

```
{...} @ clock_expr ;
```

More on Sequences



- **Sequences can be named**
 - Declare separately and reference by name in other sequences, properties or directives
- **Sequences can be parameterized**
 - Allows for easier reuse of sequences

```
sequence handshake (boolean req, boolean gnt) is
    (req) -> (eventually! gnt);

property all_reqs_are_gnted is always forall I in {0 to 15} :
    handshake(req(I), gnt(I)) @
    rose(clk);
```

Properties



- **Properties specify relationships between boolean expressions, sequences and subordinate properties over time (temporal relationships)**
 - **Characteristic behavior of the design, protocol, block or interface over time**
- **Properties can be named**
- **Properties can be parameterized**
- **Certain operators apply only to properties**

```
property ResultAfterN (boolean start, stop; property result; const n) =  
    always ((start -> next[n] (result)) @ (posedge clk) abort stop);
```

```
    ResultAfterN (write_req, cancel, (eventually! ack), 3)
```

Directives



- **What to do with the defined property(ies)**
 - **assert**: Verify that a property holds (matches)
 - **cover**: Did a property hold (non-vacuously) at least once
 - **restrict**: Constrain design inputs using specified property (sequences)
 - **restrict_guarantee**: Same as restrict plus assert
 - **assume**: Constrain the behavior of the input signals so that a property holds
 - **assume_guarantee**: Same as assume plus assert
 - **fairness**: Used to “filter” out repeated occurrences of an event that block another event.
 - **strong fairness**: Same as fairness except it guarantees that one event does not block all others

NOTE: As a general methodology, we encourage the use of only assert and cover directives

Use of assume and restrict directives are discouraged (and unsupported for, at least the near term)

vunits



- **vunit:**
 - **Verification unit**
 - **Packages properties and directives (assertions)**
 - **Binds to a specific instance in the design (which provides the context for any design name referenced in the PSL)**
 - **Not used with embedded PSL as it is redundant information**
 - **Embedded PSL already provides an implicit binding to all instances of that module/unit**
 - **Context is implicit by the location of the PSL in the HDL source**

```
vunit <vunit-name> ( <instance-name> ) {  
    default clock = @posedge (clk);  
    // default clock is @rising_edge (clk);  
  
...  
}
```

vunit Binding by Name



```
module Queue( ....);
input qClk, qInsert, qRemove;
input [31:0] qDataIn;
output qFull, qEmpty, qError;
output [31:0] qDataOut;

parameter qSize = 8;

reg [31:0] Q[0:qsize-1];
integer first, last;
wire full, empty;

assign full = (((last+1) % qSize) == first);
assign empty = (last == first);
assign qError = (qInsert && qFull) ||
                (qRemove && qEmpty);
assign qFull = full;
assign qEmpty = empty;
```



```
.....
vunit QueueRules (Queue) {
    default clock = (posedge qClk);

    assert never
        {(qFull && qInsert); qEmpty};

    assert never
        {(qEmpty && qRemove); qFull};

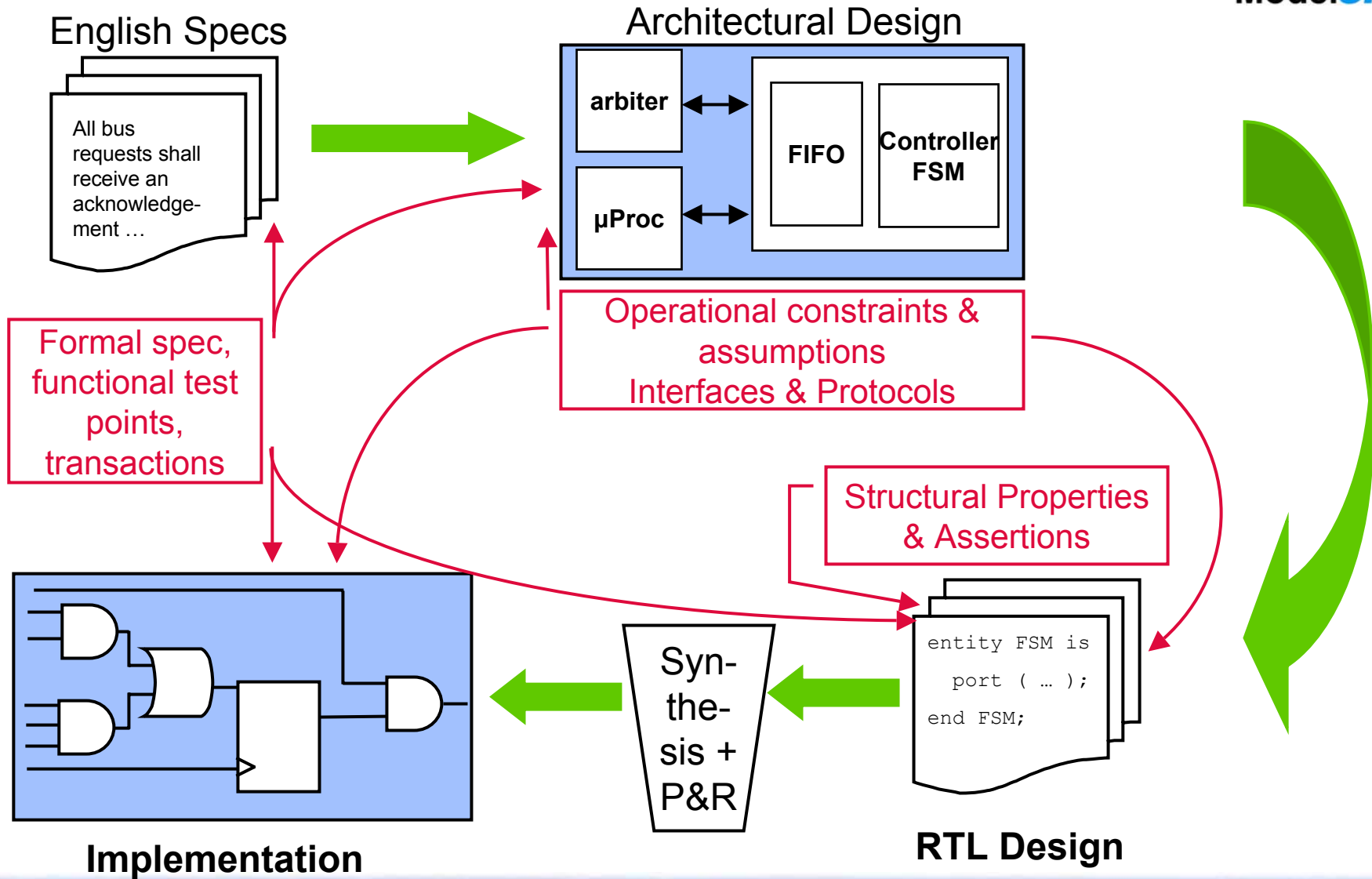
    assert never
        (qInsert && qRemove);
}
.....
```

PAV & ABV Methodology



- **Property-assisted Verification (PAV) uses properties in multiple facets of verification:**
 - Formal specification
 - Architectural responsibilities and relationships
 - Interface protocols and timing
 - Assertion-based verification of RTL design
 - Verifying a property holds (or never holds)
 - Testbench properties (stimulus generation)
 - Legal (and illegal) transactions and sequences
 - Input constraints
 - Input assumptions (especially within a context)
- **Benefits are commensurate with use**
 - Can start with ABV for RTL design
 - Can extend to other areas as comfort grows

PAV in Traditional Flow



Formal Specification



- **Weaknesses of natural language specifications**
 - **Ambiguity**
 - Including the ease of misinterpreting
 - **Cannot be executed to verify**
 - **Completeness**
 - **Accuracy**
- **PSL is a specification language**
 - **Unambiguous**
 - **Precise semantics**
 - **No emotional (connotation) baggage**
 - **Can be used to create executable specification**
 - **Coverage of properties (functionality, tests)**
 - **Accuracy of specification**
 - **Jumpstart development of Verification Environment**
 - **Legal input sequences**
 - **Input constraints and assumptions**
 - **NOTE: Probably captured in vunits (separate PSL file)**

Specification Properties



- **Functional**
 - Purpose of design
 - How it is intended to behave
- **Performance**
 - Latencies & rates
 - Frequency
 - Resources and Capacity
- **Interfaces**
 - External to design: busses, peripherals, etc.
- **Security and Safety**
 - Design behavior under specific situations
- **Operational**
 - (Re)Configuration
 - User commands
 - Maintenance
- **Note: Still using vunits and separate PSL files**

Architectural Design



- **Functional**
 - Behavior of each high level block
- **Major internal interfaces**
 - Bus protocols
 - Constraints and assumptions of block inputs
 - Required timing constraints/relationships between blocks
- **Resource sharing arbitration**
- **For block-level verification**
 - Use properties to guide stimulus generation
 - Input sequences, constraints and assumptions
- **NOTE:**
 - Specification properties continue to be applied during architectural design verification

Structural (RTL Design) Properties



- **Designers write assertions to verify**
 - Algorithms
 - Legal state transitions and FSM output
 - Resource utilization (e.g., FIFO large enough?)
 - Ensure all interesting sequences/cases are covered
- **Designers write assertions to communicate**
 - Assumptions on inputs to their module
 - Timing and format of outputs from their module
- **Designers write properties for stimulus generation**
 - If (constrained-) random or directed tests are insufficient
 - To help direct adequate code coverage for the module
- **NOTES:**
 - This is what most people consider assertion-based verification
 - The designer probably embeds the PSL directly in the HDL
 - Use Specification and Architectural properties for integration verification

Implementation (Gate) Properties



- Typically, no new properties or assertions
- Gate-level verification reuses existing properties:
 - Specification level
 - Architectural level
 - Any structural level properties that need continued verification
 - Internal state
 - Timing (cycles) along sensitive sub-paths
 - Properties being verified must:
 - Reference primary I/Os
 - Reference objects preserved through the synthesis/P&R process
- Stimulus generation properties can be reused
- Functional coverage can be used
 - Full timing gate sims typically sample the breadth of RTL tests
 - Coverage can recommend which set of tests is sufficient

ModelSim and ABV



- **ModelSim supports ABV with PSL today**
 - VHDL flavor only in 5.8
 - Verilog flavor in 6.0 (available now!!)
 - Embedded and separate files
 - SystemVerilog assertions post 6.0 release
 - Various language limitations
 - But can be used for most single clock, synchronous checks
 - Simple subset (for simulation) only
 - Assertion browser
 - TCL assertion * commands
 - Tracing of assertions in wave window
 - Currently, results are not recorded in WLF as assertions
 - Signal traces are in WLF if assertion is added to wave

Using PSL with ModelSim



- **Embedded PSL: HDL source contains PSL.**
vcom/vlog automatically compile the embedded PSL
 - vcom dram_controller.vhd
vlog dram_controller.v
 - To disable compilation of embedded PSL:
 - vcom `-nopsl` dram_controller.vhd
 - vlog `-nopsl` dram_controller.v
- **Separate PSL file: Compile with HDL source containing module it is bound to:**
 - vcom `-pslfile protocol.psl testbench.vhd`
 - vlog `-pslfile protocol.psl testbench.v`
- **vsim switch to disable assertions during simulation**
 - vsim `-nopsl tb`

Simulator Assertion Commands



- **TCL commands to enable/disable assertions, etc.**
 - Same capabilities you can do via the GUI
 - Examples:

vsim> view assertions

vsim> assertion fail [-recursive] [-enable] [-disable]
[-action continue | break | exit]
[-limit <count> | none]
[-log on | off] <path> ...

vsim> assertion pass [-recursive] [-enable] [-disable]
[-limit <count> | none]
[-log on | off] <path> ...

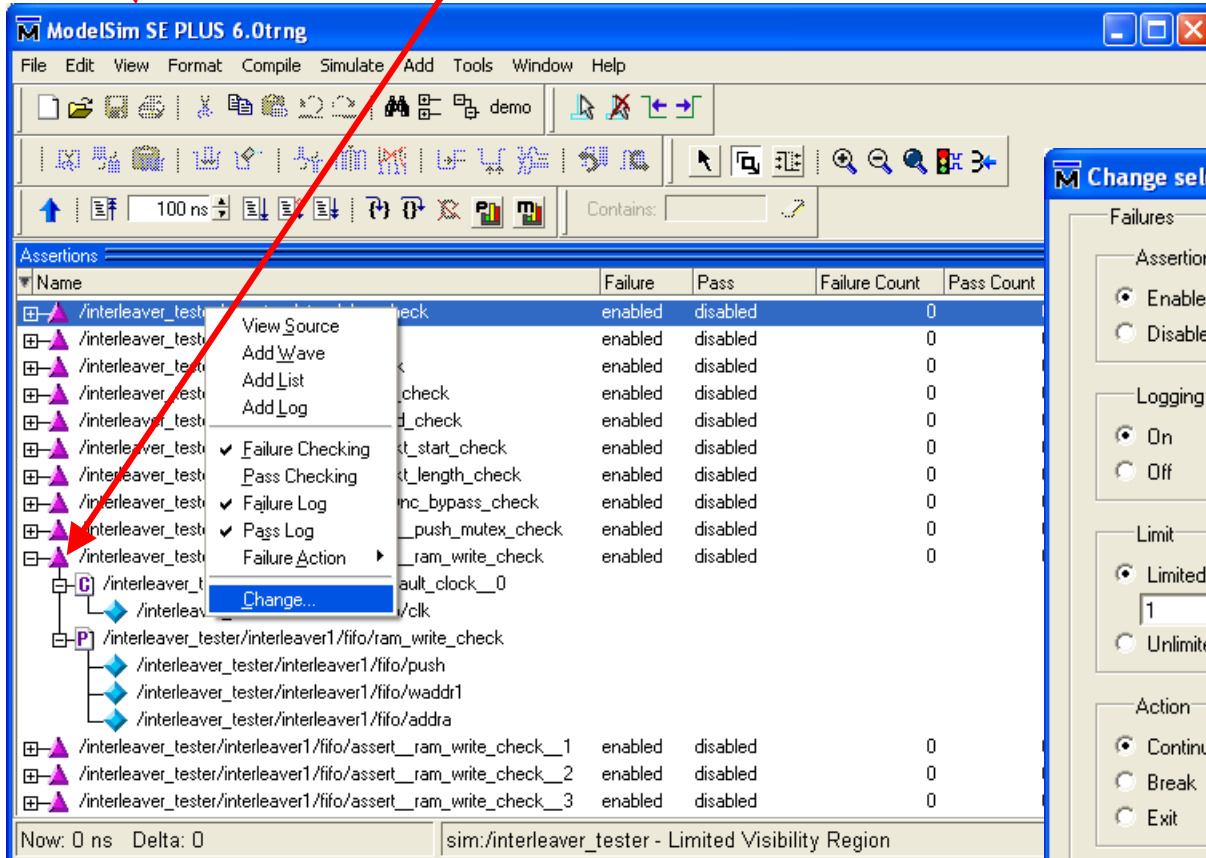
vsim> assertion report [-number] [-recursive] [-tcl_list]
[-verbose] <path> ...

New Assertion Windows

Assertion Browser

Expanded Assertion

Change Settings



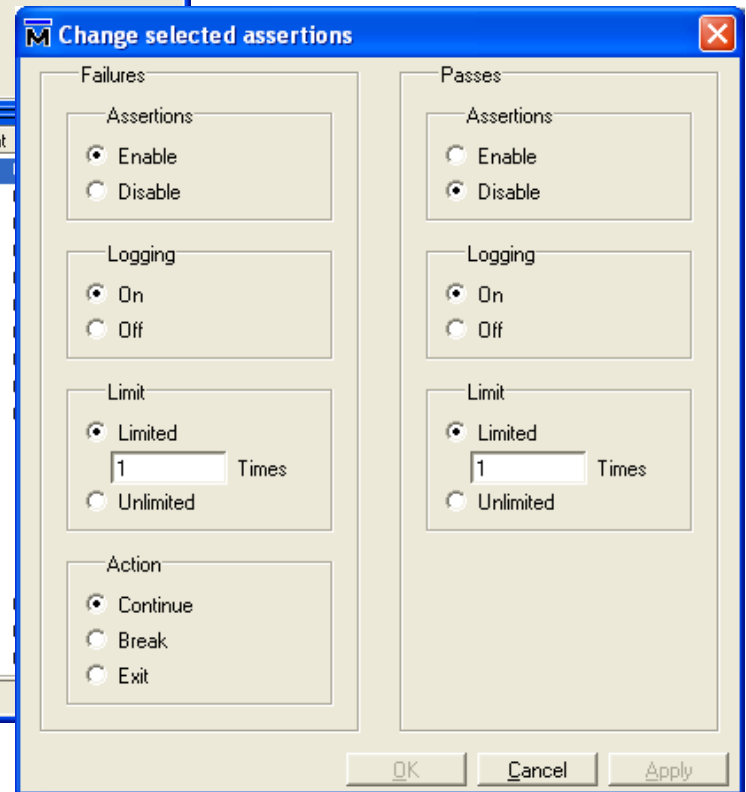
ModelSim SE PLUS 6.0trng

File Edit View Format Compile Simulate Add Tools Window Help

100 ns

Name	Failure	Pass	Failure Count	Pass Count
/interleaver_tester/interleaver1/fifo/ram_write_check	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/push	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/waddr1	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/addr1	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/assert_ram_write_check__1	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/assert_ram_write_check__2	enabled	disabled	0	0
/interleaver_tester/interleaver1/fifo/assert_ram_write_check__3	enabled	disabled	0	0

Now: 0 ns Delta: 0 sim:/interleaver_tester - Limited Visibility Region



Change selected assertions

Failures

Assertions

Enable
 Disable

Logging

On
 Off

Limit

Limited
1 Times
 Unlimited

Action

Continue
 Break
 Exit

Passes

Assertions

Enable
 Disable

Logging

On
 Off

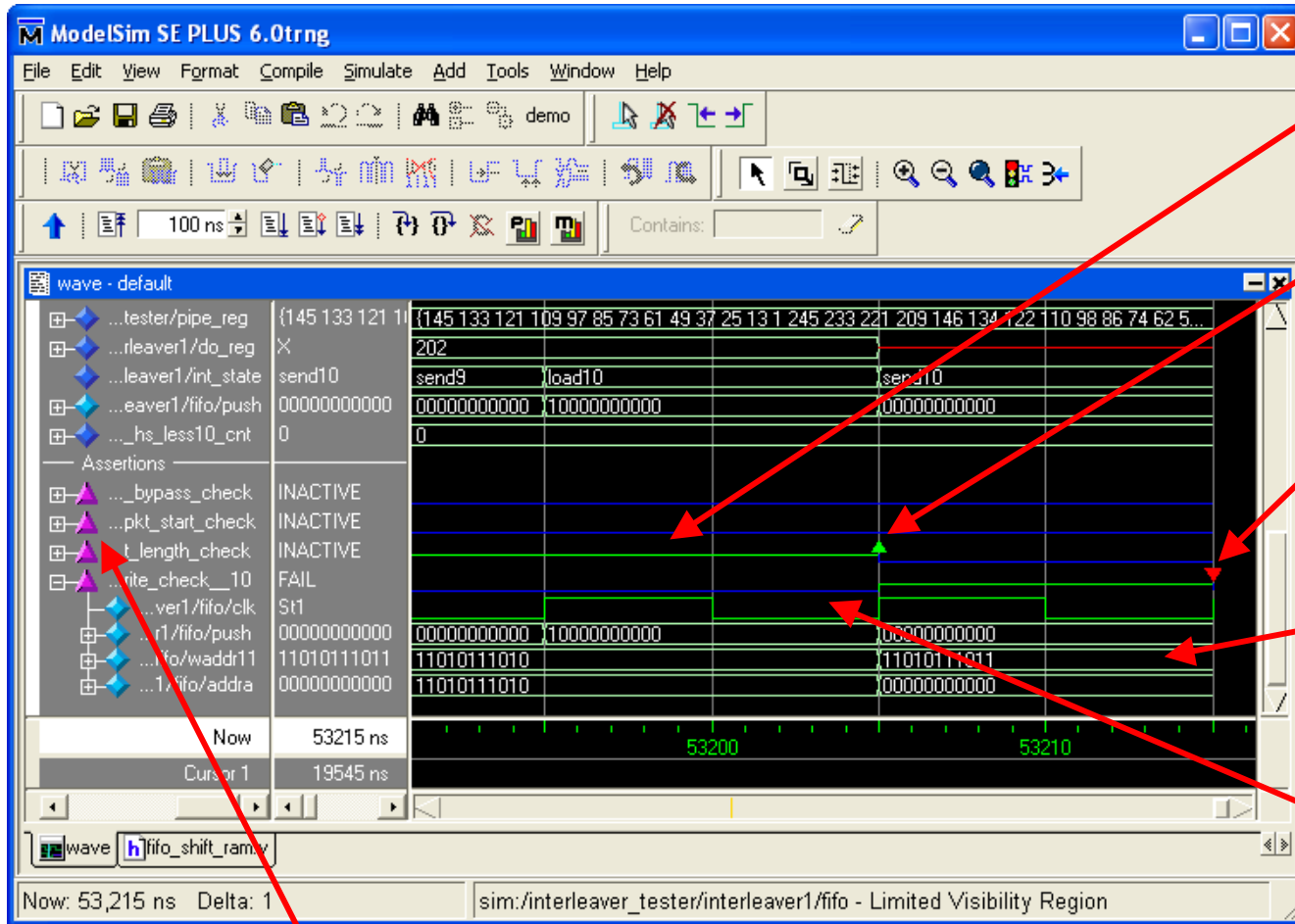
Limit

Limited
1 Times
 Unlimited

OK Cancel Apply

Add to wave window from assertion window

Assertions in Wave Window



Green mid-line indicates assertion is active

Green triangle indicates assertion passed

Red inverted triangle indicates assertion failure

Right click on waddr11 signal in wave window to bring up Dataflow window

Blue low-line indicates assertion is inactive

Simply D&D Assertions from Assertion Browser into Wave Window to view assertions
 Assertions can be expanded to view all signals associated with the assertion

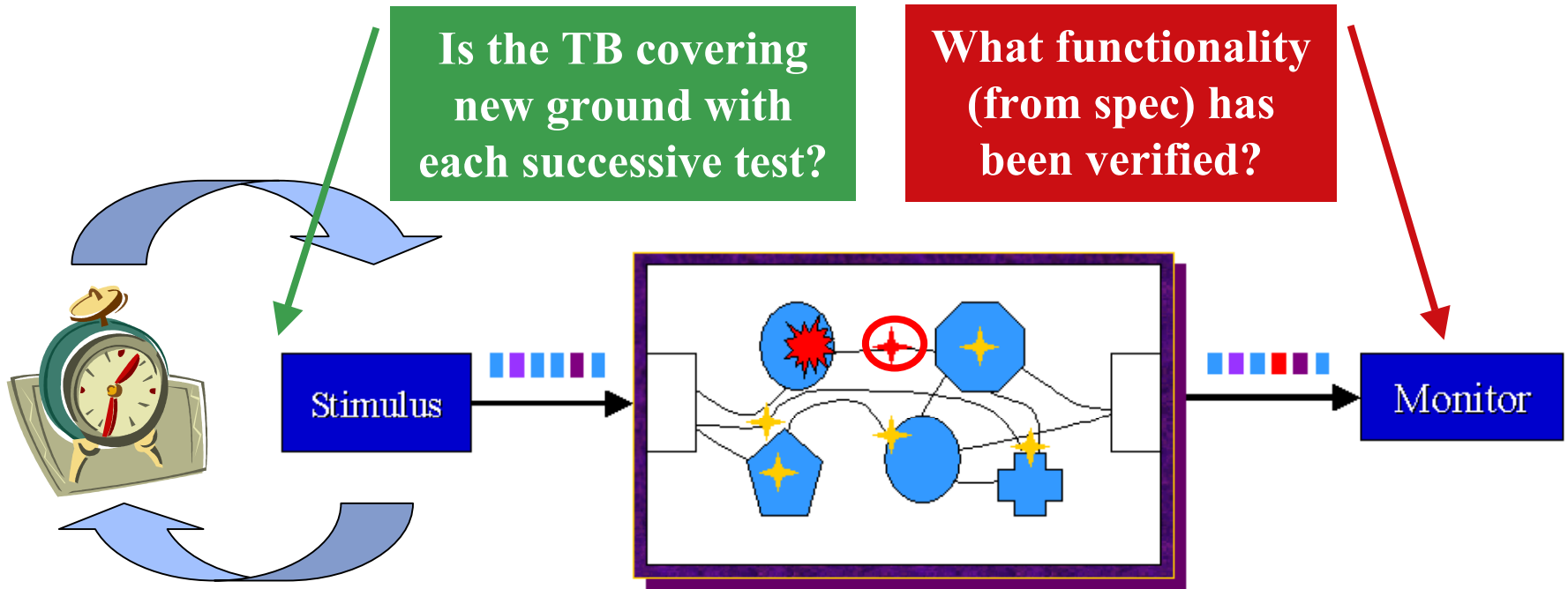
PSL Debug



```
ModelSim SE PLUS 6.0trng
File Edit View Format Compile Simulate Add Tools Window Help
demo
100 ns
Contains:
fifo_shift_ram.v
ln #
197     waddr10 <= 11'd1280;
198     else
199     waddr10 <= waddr10 + 11'd1;
200     default:
201     if (BUG == 0)
202     if (waddr11 == 11'd1722)
203     waddr11 <= 11'd1536;
204     else
205     waddr11 <= waddr11 + 11'd1;
206     else
207     if (waddr11 == 11'd1724)
208     waddr11 <= 11'd1536;
209     else
210     waddr11 <= waddr11 + 11'd1;
211     endcase
212
213 // the read address pointers needs to increment each
214 // time the write pointers are incremented. The ram read
215 // are initialized to the write address plus 1. Check for
wave fifo_shift_ram.v
Now: 53,215 ns Delta: 1 sim:/interleaver_tester/interleaver1/fifo/#ALWAYS#133,219 - Limited Visibility Region
```

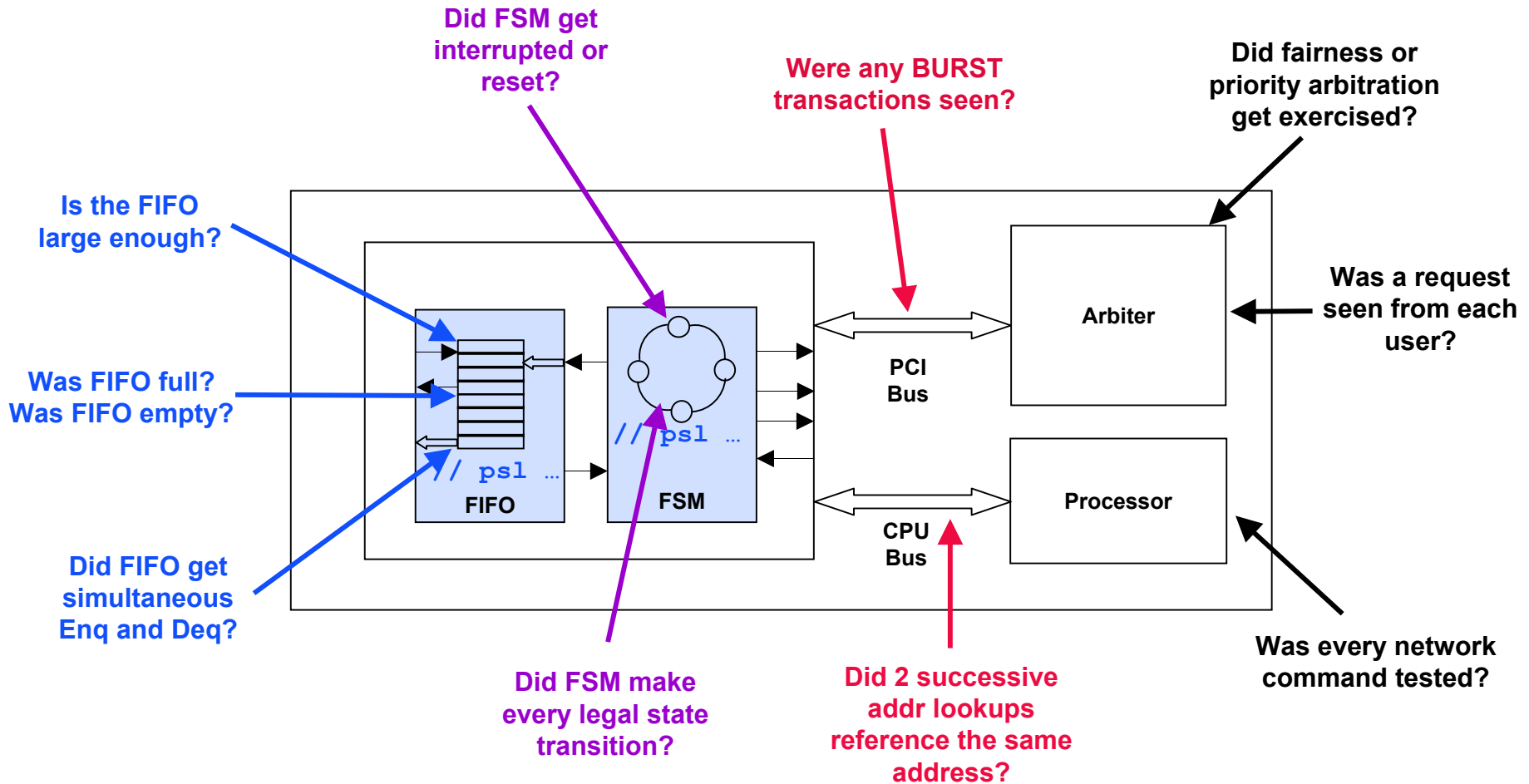
Assertions find failure => Use GUI to find source of failure

Functional Coverage



- What are you measuring?
 - Code coverage != functional coverage
- How good are your tests?
 - Ensure new tests advance towards coverage goals
- When are you done?

FC Questions



Functional Coverage



- **Properties and sequences are user-defined metrics**
 - Functional requirements specification
 - Interface specifications
 - Test plans
- **Complementary to Code Coverage**
- **Use metrics appropriately**
 - Code coverage for block/module verification
 - Easier to exercise all branches, conditions, etc.
 - Answers: Is my block/module ready for integration?
 - Functional coverage for (sub)system verification
 - Answers: Does my (sub)system work as specified?
 - Both answer: Am I done verifying at this level?

ModelSim and Functional Coverage



- **ModelSim supports Functional Coverage with PSL (6.0)**
 - PSL cover directive
 - Based on PSL sequences not properties
 - Sequence is counted if it completes
 - PSL endpoints
 - PSL construct with HDL boolean signal semantics
 - Can be used in always block or process block
 - sensitivity list
 - HDL if statement
 - Functional Coverage browser
 - Merging/saving/reloading of FCDB's
- **SystemVerilog coverage post 6.0**

Functional Coverage Browser



ModelSim SE PLUS 6.0

File Edit View Format Compile Simulate Add Tools Window Help

100 ns

Functional Coverage

Name	Enabled	Log	Count	AtLeast	Weight	Cmplt %	Cmplt graph	Included
interleaver_m0						88%		
/interleaver_tester/interleaver1/c_interleaver		Off	88	100	4	88%		✓
interleaver_tester						74%		
/interleaver_tester/c_no_dwn_hs_more_10		Off	75	100	4	75%		✓
/interleaver_tester/c_no_dwn_hs_less_10		Off	462	250	1	100%		✓
/interleaver_tester/c_no_up_hs_more_11		Off	62	100	4	62%		✓
/interleaver_tester/c_no_up_hs_less_10		Off	395	250	1	100%		✓

Total Coverage: 79%

Now: 45 us Delta: 2

sim:/interleaver_tester

Configure selected cover directives

Log

On
 Off

Counting

Enable
 Disable

Inclusion

Include
 Exclude

Set Weight to

Set AtLeast count to

OK Cancel

FC in Wave Window

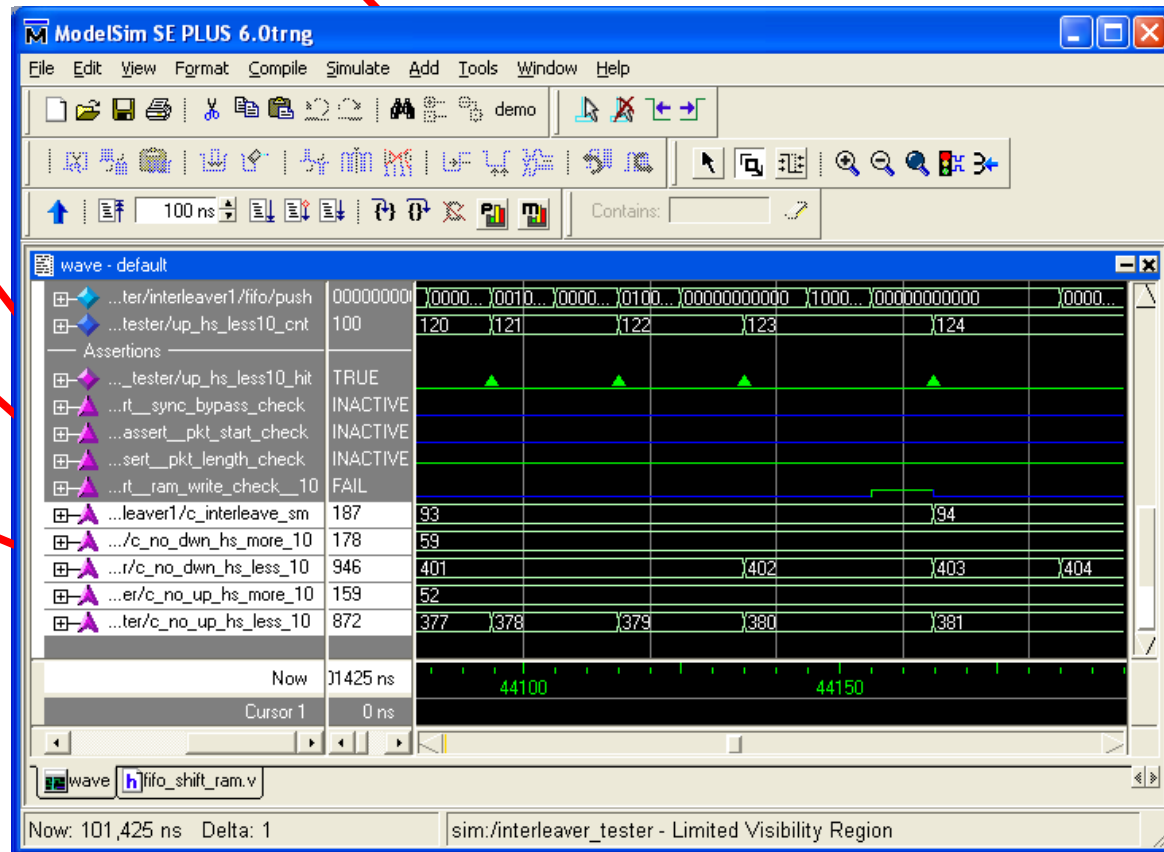
PSL Endpoint
Purple Diamond
(introduced 6.0)

PSL Assertion
Purple Triangle
(introduced 5.8)

PSL Coverage Point
Purple Chevron
(introduced 6.0)

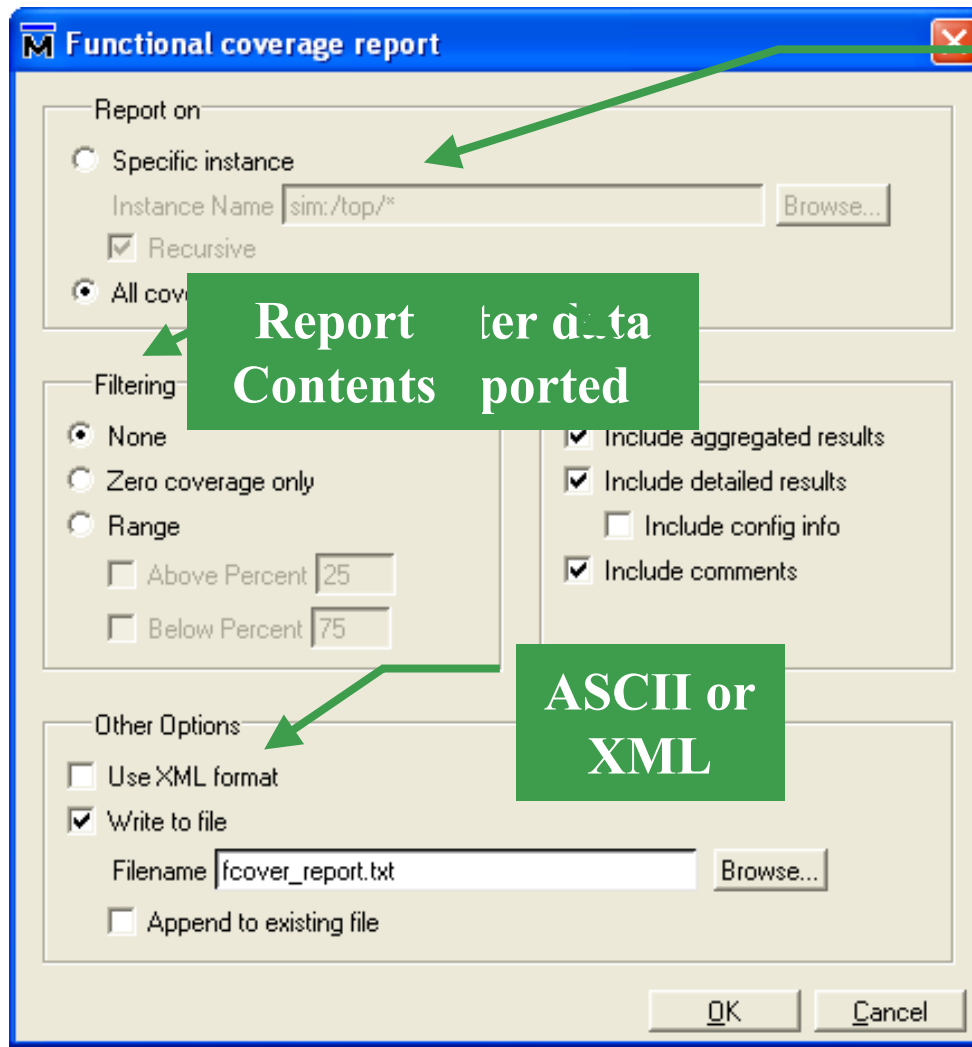
PSL Boolean
Endpoint
Asserted

Functional
Coverage Point
Temporal mode



Green Triangle
indicates
Functional
Coverage
Point Hit

Reporting FC



Specify Hierarchy Level Starting Point

Report Contents reported

ASCII or XML

- From simulator command line
- Or from vcover utility

Simulator Functional Coverage Commands



■ Functional Coverage TCL commands

- Same capabilities you can do via the GUI
- Examples:

```
vsim> fcover configure -enable|-disable] [-recursive]  
[-at_least <positive integer>
```

```
vsim> fcover report [-above <pct>] [-below <pct>] [-[no]aggregated]  
[-append] [-[no]comments]  
[-config] [-[no]details] [-du <name>] [-file <filename>]  
[-recursive] [-tcl|-xml] [-zero] <fcp> ...
```

```
vsim> fcover reload [-install <path>] [-merge] [-strip <n>] <filename>...
```

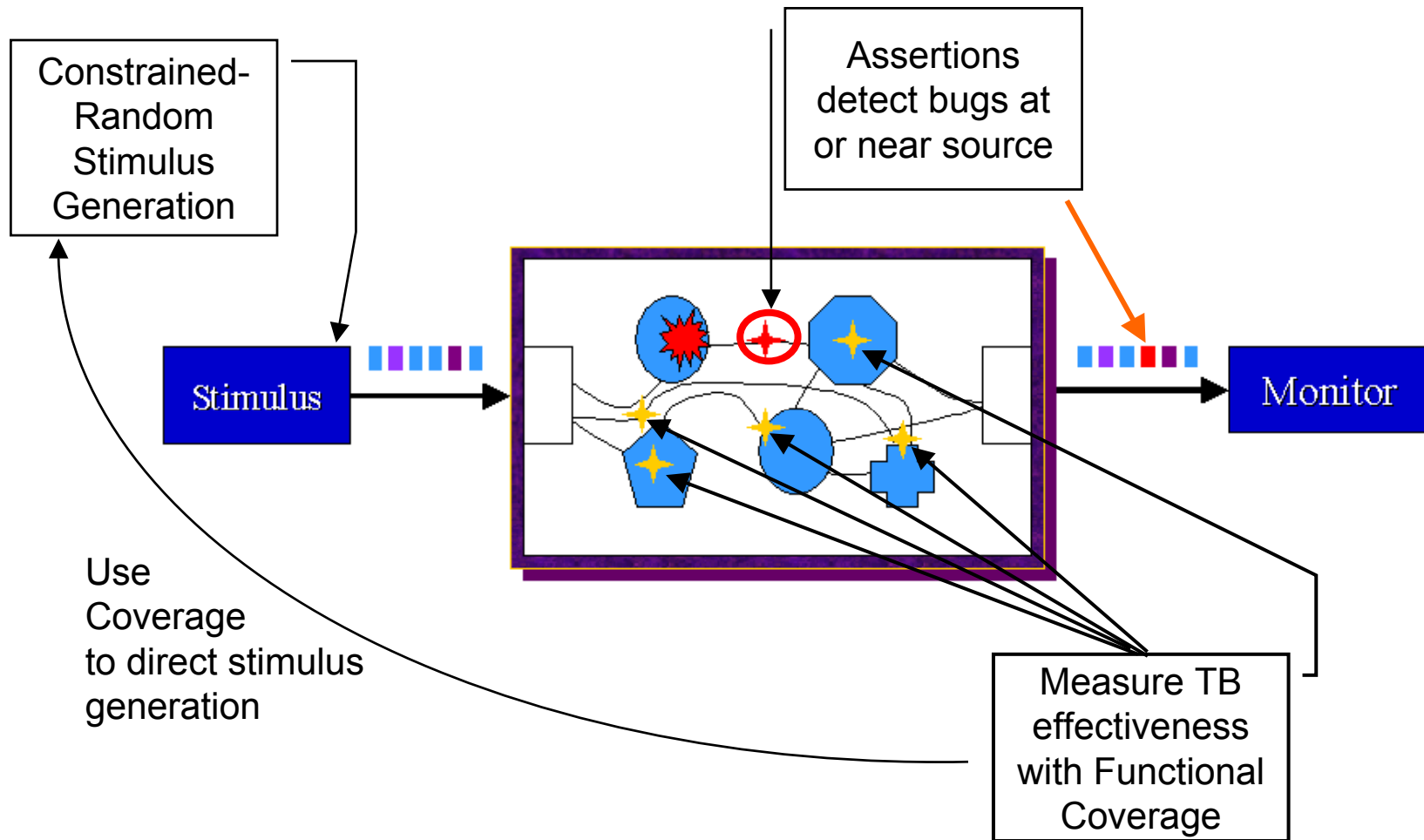
```
vsim> fcover save <filename>
```

General PSL Coding Guidelines



- **Use the always/never operator in the property not in the assertion**
 - ModelSim will preserve the property name you supply
 - `assert property mutex is always (not(a and b));`
 - ModelSim will supply its' own name: `assert_n`
 - `Property mutex is not(a and b);`
 - `assert always mutex;`
 - Directive references named property
- **Do not use implication with never**
 - Almost always (nearly 100%) you will not get what you think you want if you use implication with never
- **Keep sequences and properties as simple as possible**
 - Build complex properties out of simple, short sequences
- **Limit large number of 'in-flight' checks simulator has to keep track of**
 - Keep assertions synchronous
 - Try to avoid triggering the same assertion check every cycle for multi-cycle signals

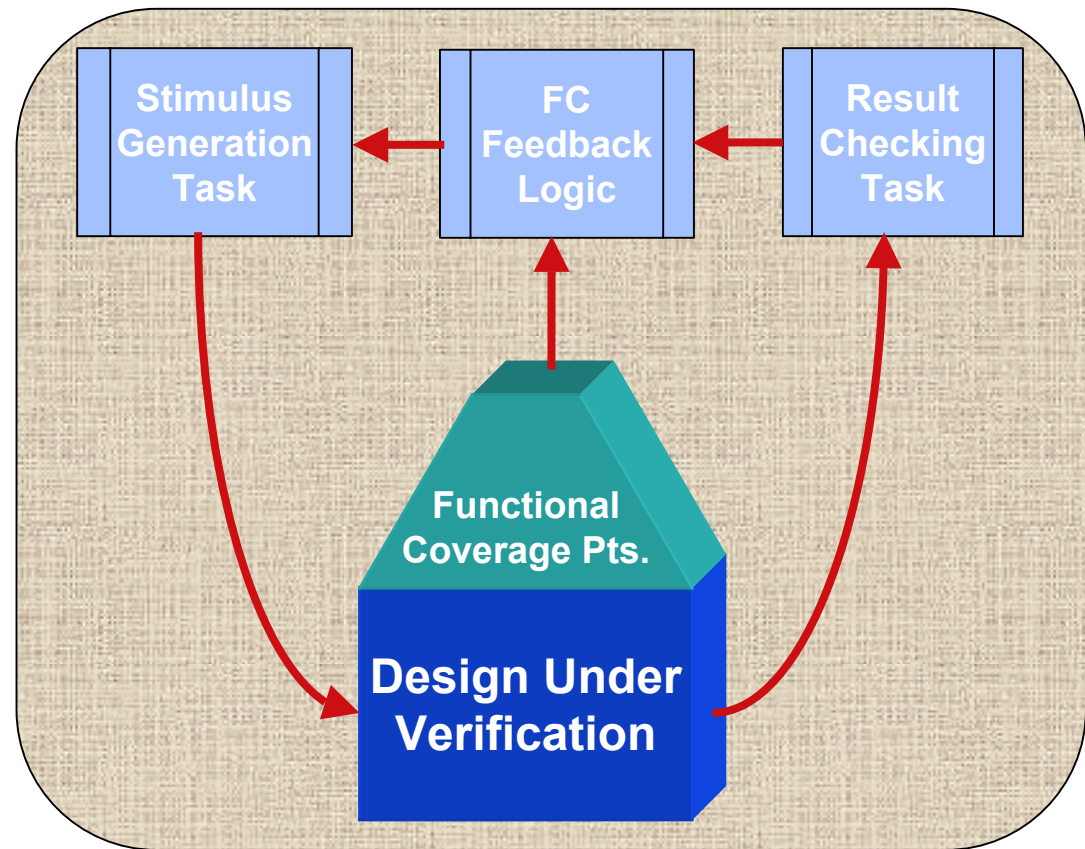
Closing the Verification Loop



Coverage-Driven Verification



- Coverage-driven verification integrates
 - Functional coverage
 - Stimulus generation
- Makes efficient use of verification resources
 - Guide stimulus generation away from covered areas towards uncovered areas
 - Minimize hand-created stimulus



Roadmap for CDV



- **The longer term roadmap also includes:**
 - **Property-guided stimulus generation**
 - **Synergism with functional coverage**
(closing the coverage/stimulus loop)
 - **And more**
 - **Implementation will follow SystemVerilog**
 - **Watch this space for future updates**
- **Verification automation is strategic**
 - **Additional R&D headcount**
 - **Required for SystemVerilog as well**

PSL Resources



- **PSL LRM:**
 - <http://www.accellera.org/PSL-v1.1.pdf>
- **Using PSL/Sugar for Formal and Dynamic Verification**
 - Book by Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari
 - 2nd Edition
 - Foreword by Stephen Bailey
 - www.vhdlcohen.com
- **Training Resources**
 - Mentor's Training Group
 - Doulos
 - Ben Cohen
 - WillametteHDL

Summary

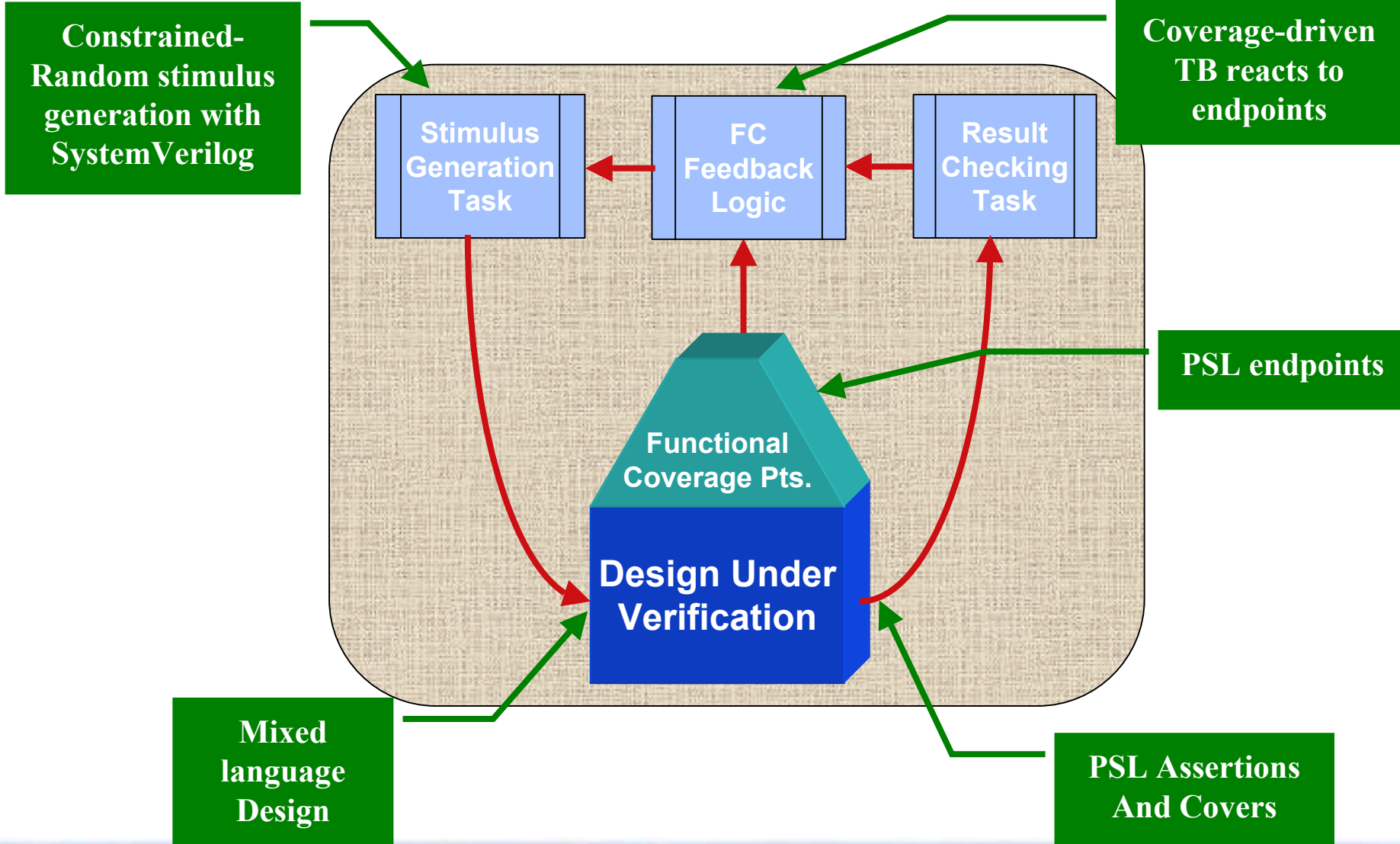


- **Improve your Verification Productivity & Methodology with**
 - Assertion-Based Verification
 - Functional Coverage
 - Coverage-Driven Verification
- **ModelSim delivers today with PSL**
 - Verilog/VHDL support
 - Assert and cover directives
 - Endpoints
 - Pre-defined library of PSL based checkers
- **Coming soon with SystemVerilog Assertions**
- **ModelSim extends its leadership in providing the best verification capabilities across all languages**
- **Mentor is the only EDA vendor supporting all Standards**

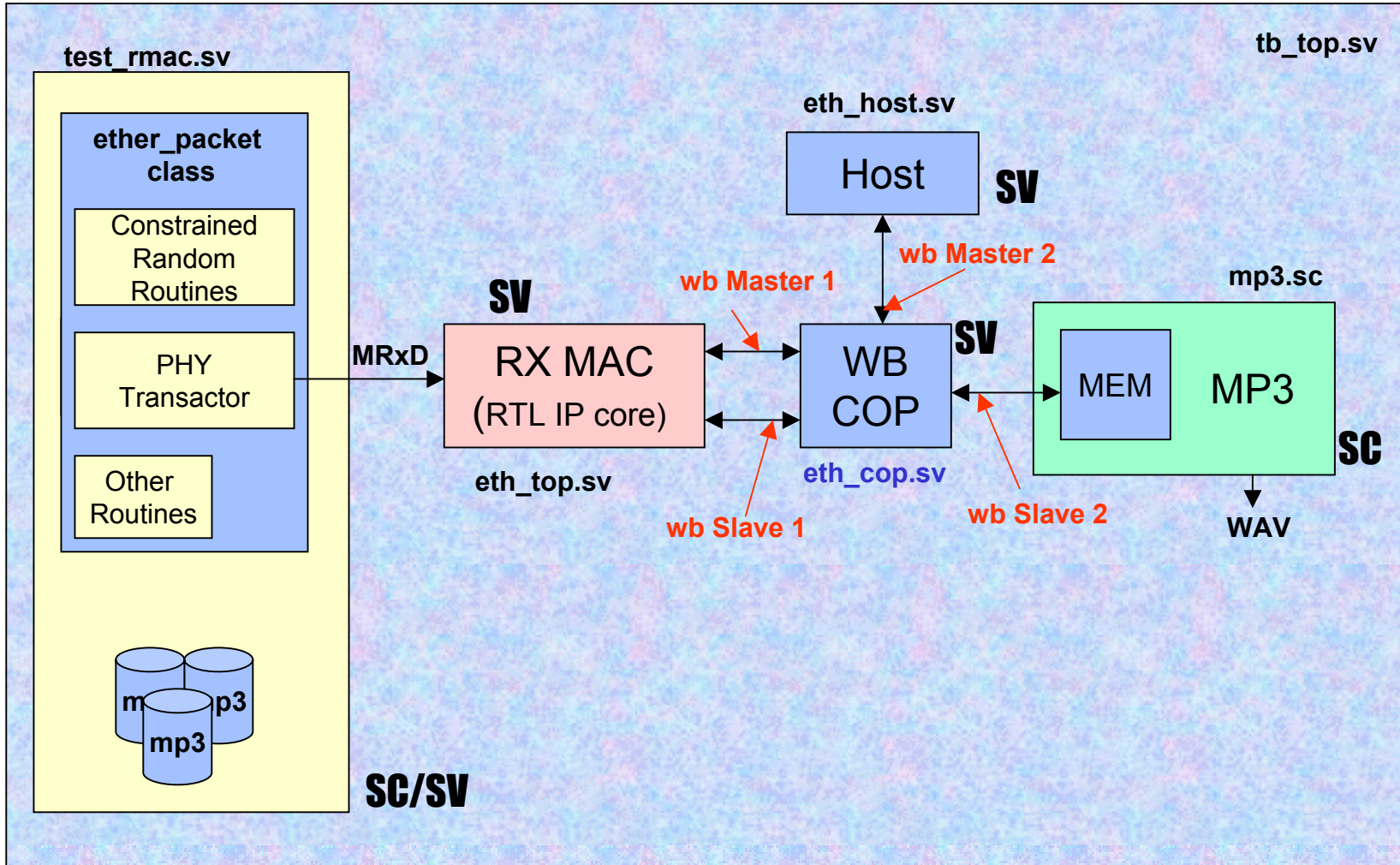
Ethernet MP3 Demo



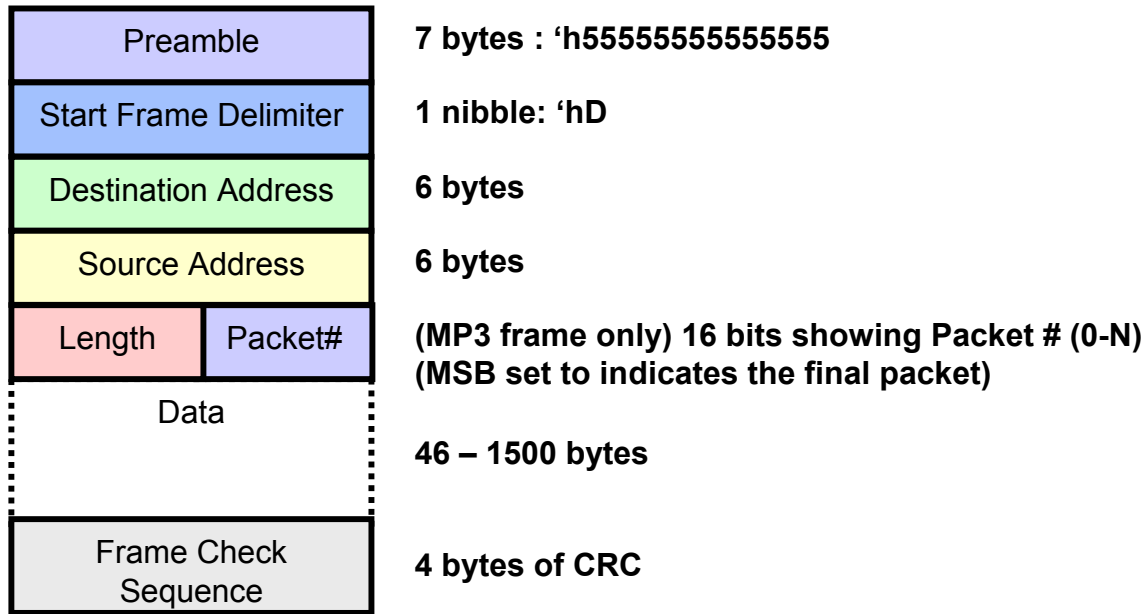
ModelSim®



Demo Circuit



Randomization of Ethernet Frame



- Multiple levels of randomization:
 - Randomize size of the packet we're sending
 - Randomize type of payload - valid data and noise
 - Randomize packet distribution across the ethernet

PSL Advantages



- **Industry standard**
 - Portability
 - Broad tool support
 - Competition on best solution
 - Verification IP availability
- **General property specification language**
 - Can specify an infinite number of checks
 - Not limited to what is predefined by a tool or library of checkers
- **HDL-neutrality**
 - PSL defined to work with both VHDL and Verilog
 - Also general description language allows PSL to work with any language (e.g., SystemC)
- **Future standards benefits**
 - VHDL IEEE working group plans to include PSL as part of VHDL
 - Accellera has directed harmonization of PSL and SystemVerilog Assertions



PSL vs OVL



- **OVL is a library of predefined checks**
 - Parameterized HDL checker modules
 - Limited number of checkers
- **OVL is not a language**
 - Not functionally complete
 - Must cobble together multiple OVL checker instances
 - Perhaps with intermediate signals
 - Or write your own in HDL
- **Requires verification engineers to edit RTL code**
 - No support for PSL's external file capability – vunits
- **Performance**
 - HDL used as property specification
 - Requires extensive analysis to optimize as checkers
- **PSL version of OVL is complementary to PSL**
 - Provides a predefined library of basic checkers

Why Not Use a HDL?



- **PSL is a specification language**
 - What, not How
 - Conciseness
 - How requires significantly more verbosity
 - Easier to optimize as context and use is understood
 - Performance
- **Compare PSL and HDL versions of same checker**

Verilog Transition Checker



```
module assert_transition (clk, reset_n, test_expr, start_state, next_state);
  input clk, reset_n;
  input [width-1:0] test_expr, start_state, next_state;
  parameter INIT_STATE=0;           parameter CHECK_STATE=1;
  reg assert_state;                 initial assert_state = INIT_STATE;
  integer error_count;             initial error_count = 0;

  always @(posedge clk) begin
    if (reset_n != 1'b0) begin
      case (assert_state)
        INIT_STATE:
          if (test_expr == start_state)
            assert_state <= CHECK_STATE;
        CHECK_STATE:
          if (test_expr != start_state) begin
            assert_state <= INIT_STATE;
            if (test_expr != next_state) begin
              ovl_error("");          // NOTE: Task defined elsewhere
            end
          end
      endcase
    end
    else begin
      assert_state <= INIT_STATE;
    end
  end // always
endmodule
```

PSL Transition Checker



```
module assert_transition (clk, reset_n, test_expr, start_state, next_state);
    input clk, reset_n;
    input [width-1:0] test_expr, start_state, next_state;

    // psl property ASSERT_TRANSITION =
    //   always ((reset_n != 1'b0) ->
    //       (always{(test_expr == start_state)} | =>
    //           {(test_expr == start_state) || (test_expr == next_state)}))
    //       abort (`ASSERT_AL_RESET == 1'b0))
    //   @(posedge clk);
    // psl assert ASSERT_TRANSITION;

endmodule
```

VERILOG: 26 lines

PSL: 11 lines

And this is a relatively simple checker!

PSL vs System Verilog Assertions



- **SVA viable solution for Verilog-only users**
 - VHDL and SystemC users must use something else
 - Or use mixed-HDL simulation with non-standard SVA support
 - VHDL to incorporate PSL as property specification capability
 - PSL's language neutrality allows use in mixedHDL designs
 - HDL neutrality improved in 1.1 LRM
 - Further improvements anticipated in the future
 - HDL-neutrality important for IP providers

- **Otherwise, SVA and PSL have similar capabilities**
 - PSL specification capabilities slightly richer
 - Temporal branching logic (formal verification)
 - Asynchronous/unclocked properties

PSL vs OVA or Other Proprietary Solutions



- **OVA is not a standard**
 - Synopsys proprietary based solution
 - Limited industry support
 - Limited IP availability
 - Synopsys says OVA is same as SVA
 - Not!
 - If it were, then why call it something else?

- **0-in**
 - Acquired by Mentor
 - Integration to ModelSim ABV methodology is TBD
 - 6.1 release at the earliest
 - Provide Verification IP
 - Formal Verification capabilities
 - Static and Dynamic
 - Currently support PSL and SVA (3.1)