



IEEE P1800 SystemVerilog Committee Meeting



February 20, 2007

SystemVerilog Enhancement Proposal Outline

Checker Libraries and Formal Verification

Dmitry Korchemny, Intel Corp.

Outline

- Introduction
- Motivation
- LTL
- Immediate assertions
- Compiler directives
- Meta-language layer
- Modeling
- Checkers
- Proposal at a glance

SystemVerilog as Assertion Specification Language

- SystemVerilog is (also) an assertion specification language
 - Citations from LRM:

SystemVerilog is a unified hardware design, **specification**, and **verification** language

SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, **specification of assertions**, coverage, and testbench verification that is based on manual or automatic methodologies.

- Assertion specification language should support checker libraries and formal verification activities
- Formal validators became users of SystemVerilog
 - Their needs and requirements differ from the needs of traditional Verilog users

Claim: SV does not have a good support of checker libraries

- Example
 - Let's write a simple library checker
 - A group of signals have the same value
 - How?
 - As an immediate assertion?
 - As a concurrent assertion?
 - As a module/interface?

“Same value”. Immediate assertion

- Cannot be put outside of procedural code

```
`define ASSERT_SAME(X, Y) assert ((X) == (Y))
```

```
module M;
```

```
    wire a, b; logic clk;
```

```
    assign a = b;
```

```
    `ASSERT_SAME(a, b);
```

✗

```
    always @(posedge clk)
```

```
        `ASSERT_SAME(a, b);
```

✓

```
endmodule
```

“Same value”. Concurrent assertion

- Cannot be put within a for-loop or within a function
- Always requires explicit clock specification

Always required. Even if may be inferred from context

```
`define ASSERT_SAME(X, Y, CLK) assert property (CLK (X) == (Y))
...
always @(posedge clk) begin
    ...
    `ASSERT_SAME(c, d, posedge clk);           ✓
    for (int i = 0; i < 8; i++) begin
        ...
        `ASSERT_SAME(a[i], b[i], posedge clk);  ✗
    end
end
end
```



“Same value”. Module/interface

- Cannot be instantiated within procedural code
 - Context sensitivity is lost
- Requires explicit width and clock specification

```
`define ASSERT_SAME(NAME, X, Y, WIDTH, CLK) lib_same \  
    NAME #(WIDTH) (X, Y, CLK)  
...  
bit [15:0] a, [15:0] b;  
  
`ASSERT_SAME(prop1, a, b, 16, clk);  
  
always @(posedge clk) begin  
    ...  
    `ASSERT_SAME(prop2, a, b, 16, clk);  
end
```

Redundant

✓

✗

Modules/interfaces are the only option to package several assertions together

“Same value”. Repeatable arguments

- No support for repeatable arguments

- Impossible:

```
`ASSERT_SAME(a, b, c);
```

- Neither macros, nor properties will help
 - Neither of them can have variable number of arguments

Property library. Expressive power

- LTL is not part of the language
 - Implementation using sequences is often tricky
 - Try to implement *eventually always a*
 - Solution:
 - » *not (##[1:\$] a |-> ##[1:\$] !a)*
 - Main problem: Checkers are not extendable

Cannot be
a sequence

SVA

```
property between(start_ev, end_ev, cond);  
    start_ev ##0 ! (end_ev && cond) [*1:$] |-> cond;  
endproperty : between
```

SVA
+ LTL

```
property between(start_ev, end_ev, cond);  
    start_ev |-> cont until (end_ev and cond);  
endproperty : between
```

Can be
a
sequence

Proposal. LTL layer

- LTL operators
 - always
 - eventually
 - until
- Distinguish between weak and strong versions
- Reset operators
 - accept_on
 - reject_on



Immediate assertions

- Immediate assertions are needed both in dynamic and formal verification
 - E.g., only immediate assertions are allowed within functions
 - Even if we make concurrent assertions legal there, they will be almost useless
- Main issue:
 - Immediate assertions may exhibit glitch behavior during one simulation tick resulting in false negatives
 - ☹ Immediate assertions are problematic for ABV



Proposal: final assertions

- Glitch-free immediate assertions

```
assert final (a) else $error ("Failed");
```

- Checked as immediate assertions, but
 - Action blocks are executed in the reactive region according to the latest check at each call site
 - Loops with statically computable bounds are unrolled
- May be written outside of procedural code
- Support *assume* and *cover* statements for final assertions

Compiler directives

- At present macros are almost useless for assertions
 - No variable number of arguments, no conditional definition

```
property trig (a, b, clk = gclk, rst = 0); //...  
`define ASSERT_TRIG(a, b, clk, rst) ???
```

```
`define ASSERT_ONE_HOT(a, clk) \  
  `ifdef clk  
    assert property (@(clk) $onehot(a))  
  `else  
    assert ($onehot(a))
```

x

Compiler directives (cont.)

- Common claim: compiler directives may be easily abused
 - But having some basic functionality is important for customization
- E.g., macro is the only mean to pack different kinds of assertions in a checker library

```
`define ASSERT_ONE_HOT(a) assert final($onehot(a))  
`define ASSERT_NTRIG(a, b, clk) \  
    assert property (@(clk) a | => b)
```

- Library user should just write ``ASSERT_ONE_HOT` or ``ASSERT_NTRIG`
 - Without need to distinguish between `assert` and `assert property`

 Need to enhance compiler directives

Meta-language. Elaboration time messages

- Enable custom error reporting at the elaboration stage

Initialization

```
initial begin
  if (num_cks < 2)
    ovl_error_t("Illegal value set for action_on_new_start");
end
```



Elaboration

```
generate if (num_cks < 2)
  $error("Illegal value set for action_on_new_start");
endgenerate
```

Meta-language. Let statement

- Let statement is a template for an expression
 - Similar to sequences and property
 - May be instantiated anywhere where an expression is allowed

```
let ap = $past(a);  
always @(posedge clk) b <= ap && c;
```

Function cannot
be used here

- Let is better than a macro
 - Known at the compilation time
 - Has a local scope
- May be used as a template for immediate assertions

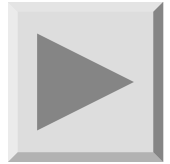
Let statement (cont.)

- Let statement is important for validators

```
wire type(memory_cluster.unit_abc.block18.base +  
  cluster3.unit_def.block145.displacement + 1) address =  
  cluster3.unit_abc.block18.base +  
  cluster3.unit_def.block145.displacement;  
...  
assert property (address < limit)
```



```
let base = cluster3.unit_abc.block18.base;  
let displacement = cluster3.unit_def.block145.displacement;  
  
let address = base + displacement;  
...  
assert property (address < limit)
```



Multiple arguments

- Q: How to handle properties (functions, etc.) having variable number of differently typed arguments?
 - Arrays cannot help
- Proposal: specification
- Proposal: argument list processing
 - generate statements with let
 - Need to allow generate statements inside a property, etc.
 - functional constructs

```
property event_order (en, *seq);
```

Multiple
arguments

```
$convolute(+, $argv)
```

= arg1 + arg2 + ...

Modeling

- Not all assertions are written using declarative SVA style. Sometimes explicit modeling is required

OVL assert_cycle_sequence

```
initial seq_queue = {num_cks{1'b0}};

always @ (posedge clk) begin
    if (`OVL_RESET_SIGNAL != 1'b1) seq_queue <= {num_cks{1'b0}};
    else begin
        seq_queue[num_cks-1] <= NC2 ? (~(|seq_queue[num_cks-1:1])) && event_sequence[num_cks-1] :
                                   event_sequence[num_cks-1];
        seq_queue[NUM_CKS_2:0] <= (seq_queue >> 1) & event_sequence[NUM_CKS_2:0];
    end
end

property ASSERT_SEQUENCE_TRIGGER_ON_FIRST_P;
    @(posedge clk) disable iff (`OVL_RESET_SIGNAL != 1'b1)
        not (&((seq_queue[num_cks-1:1] & event_sequence[num_cks-2:0]) | ~(seq_queue[num_cks-1:1]))) != 1'b1;
endproperty

property ASSERT_SEQUENCE_TRIGGER_ON_MOST_P;
    @(posedge clk) disable iff (`OVL_RESET_SIGNAL != 1'b1) seq_queue[1] |-> event_sequence[0];
endproperty
```

Modeling. Current solution

- (System)Verilog RTL layer is used for modeling
- Problems:
 - Synthesis
 - Initialization
 - Assignment loops
 - Partial assignment
 - Assignment on clock edge
 - Sampling

```
always @(posedge clk) a <= b;  
always @clk a <= b;
```



Modeling. Transition relation

- Transition relation: $R(v, v')$
 - v – current value
 - v' – next value
- There is a need in next value modeling
 - v' is the value of expression at the next tick of the system clock



- System clock should be a language construct
 - System clock = global clocking

Property packaging

- Typical OVL checker contains
 - One or more assertions
 - Modeling code
 - Several cover statements
- Such checkers can be packaeded into module/interface only
 - Not a good solution



- Special packaging units are required for checkers

Proposal at a glance

- Enhance compiler directives to make them usable for assertions
- Enhance meta-language layer
 - Expression templates
 - Multiple arguments
- Introduce global clocking to support cycle-based design
 - Necessary for next value functions
- Add LTL (temporal logic) layer, similar to PSL
- Introduce checkers – a new packaging unit for assertions and modeling code
- Define special semantics for modeling code within checkers
- Introduce final assertions as glitch-free immediate assertions

Acknowledgments

- S. Bresticker, Intel
- D. Bustan, Freescale
- E. Cerny, Synopsys
- P. Dasgupta, Synopsys
- S. Dudani, Synopsys
- J. Havlicek, Freescale
- H. Kerem, Intel
- Y. Levy, Intel
- M. Maidment, Intel
- L. Piper, Cadence
- A. Saha, Synopsys
- E. Seligman, Intel



Backup slides



Immediate assertions

```
module bar;
bit clk, m, n, o, ff1,ff2, i_ff1,i_ff2, tmp;
always_comb begin assert (m == ff1);
    case({m,ff1})
        2'b00: o = 1'b0;
        2'b11: o = 1'b1;
    endcase
end
always_comb tmp = 1; always_comb n = ff2;
always_comb m = n && tmp;
always_ff @(posedge clk) ff1 <= i_ff1;
always_ff @(posedge clk) ff2 <= i_ff2;

initial forever begin #1; clk = ~clk; end
initial begin
    @(clk) begin    i_ff1 <=1; i_ff2 <=1;    end
    #1000; $finish;
end
endmodule
```

{m,ff1} == 01 <- Fire
{m,ff1} == 11 <- Does not fire



Module-based implementation

- Property: whenever *start* signal is asserted, *ready* bit should be set

```
assert_implication ready_ok (clk, reset_n, start, ready);
```

- Modified property: whenever *start* signal is asserted, *ready* bit should be set in two phases

```
assert_implication ready_ok  
(clk, reset_n, start, ##2 ready);
```



Use
assert_cycle_sequence



Multiple argument handling

- Example
 - Summing arguments using *let* and *generate* statements
 - *\$argv* – argument list
 - *\$argv(i)* – i^{th} argument

```
for (genvar i = 0; i < $size($argv); i++) begin : b1
  if (i == 0) begin : b2
    let res = '0;
  end
  else begin : b2
    let res = b1[i-1].res + $argv(i);
  end
end
let res = b1[$size($argv) - 1].b2.res;
```

Checker example (OVL checker)

```
typedef enum { cover_none, cover_all } coverage_level;
checker assert_window (
    integral test_expr, start_event, end_event,
    event clk = $inferred_clock,
    integral reset_n = !$inferred_reset,
    const string msg = "VIOLATION";
    const coverage_level clevel = cover_all;
);
    default clocking @clk; endclocking
    default disable !reset_n;
    bit window = 0;

    function bit next_window;
        if (!reset_n || window && end_event) return 1'b0;
        if (!window && start_event) return 1'b1;
        return window;
    endfunction

    window <= next_window;

    property ASSERT_WINDOW_P;
        (start_event && !window) | => test_expr until (end_event && test_expr);
    endproperty

A_ASSERT_WINDOW_P:
    assert property (ASSERT_WINDOW_P) else $error(msg);

    if (coverage_level != ovl_cover_none) begin
        cover_window_open: cover property ( start_event && !window ) $display("window_open covered");
        cover_window:      cover property ( start_event && !window ##1
                                           !end_event && window [*0:$] ##1 end_event && window )
                           $display("window covered");
    end
endchecker : assert_window
```

Argument declaration

Modeling

Properties

Checker example. Declaration

Interface is uniform: no distinction between parameters and ports

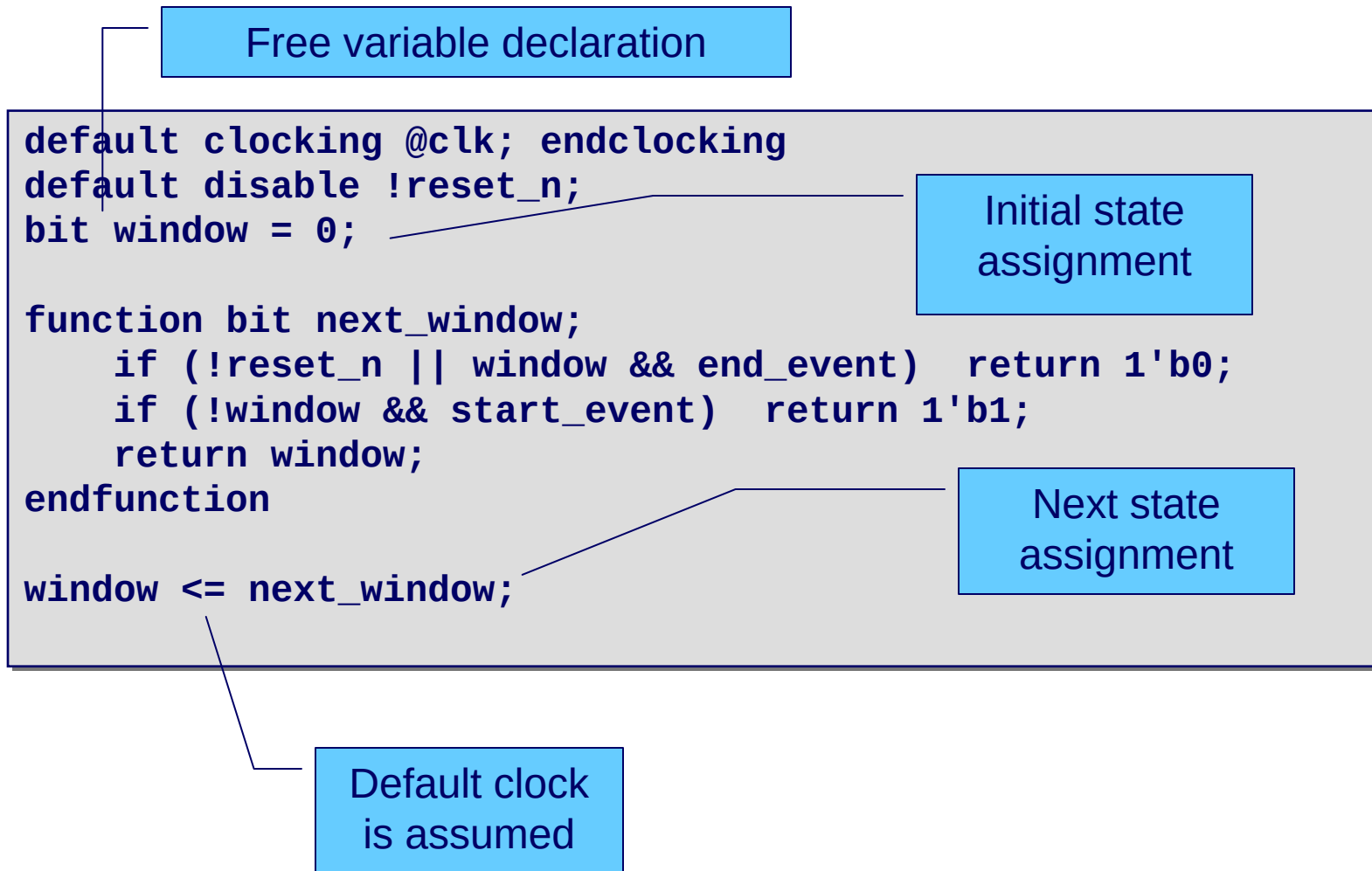
```
typedef enum { cover_none, cover_all } coverage_level;  
  
checker assert_window (  
    integral test_expr, start_event, end_event,  
    event clk = $inferred_clock,  
    integral reset_n = !$inferred_reset,  
    const string msg = "VIOLATION";  
    const coverage_level clevel = cover_all;  
);  
...  
endchecker : assert_window
```

Clock is passed
as an event,
may be inferred
from context

Integral type: neither
sequence, nor property

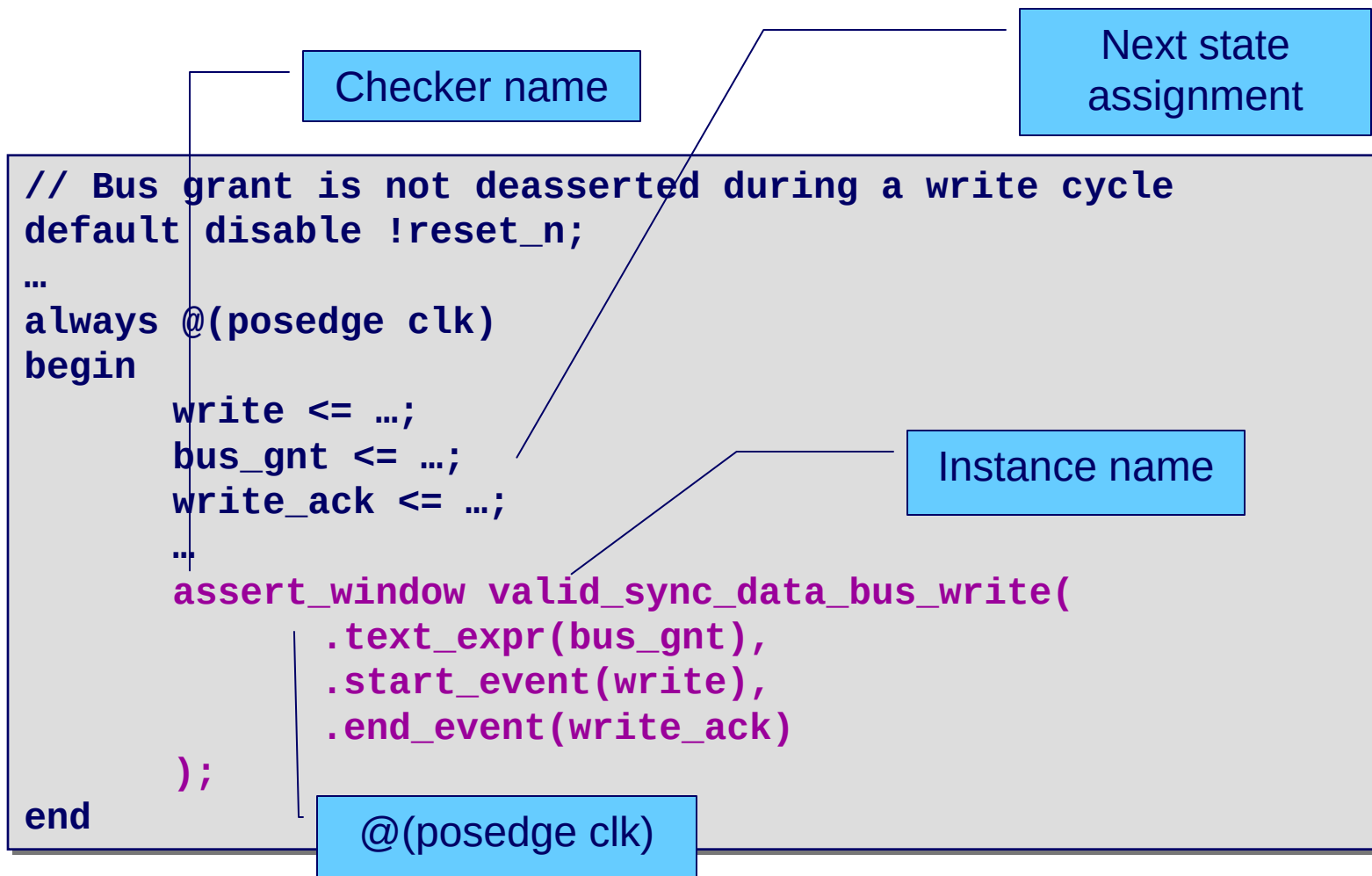
Validity checked at
elaboration time

Checker example. Modeling





Checker example. Instantiation





Final assertions

```
p1: assert final(x);  
  
always @(posedge clk) begin  
    a = b;  
    p2: assert final(a); // Instance #1  
    a = c;  
    p3: assert final(a); // Instance #2  
    for (int i = 0; i < 4; i++)  
        p4: assert (b[i]); // 4 instances  
end
```