2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SVA Advanced Topics: SVAUnit and Assertions for Formal

SystemVerilog Assertions Verification with SVAUnit

Andra Radu

Ionuț Ciocîrlan

Tutorial Topics

- Introduction to SystemVerilog Assertions (SVAs)
- Planning SVA development
- Implementation
- SVA verification using SVAUnit
- SVA test patterns

Introduction to SystemVerilog Assertions (SVAs)

Assertions and Properties

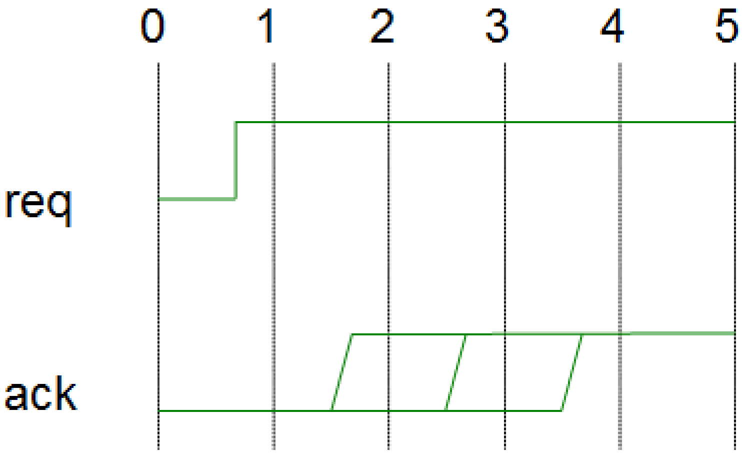
- What is an assertion?

```
assert (a |-> b)
else $error("Assertion failed!")
```

- What is a property?

```
property p_example;
 a |-> b
endproperty
```

Simple Assertion Example



After the rise of request signal, the acknowledge signal should be asserted no later than 3 clocks cycles.

```
property req_to_rise_p;
  @(posedge clk)
  $rose(req) |-> ##[1:3] $rose(ack);
endproperty
```

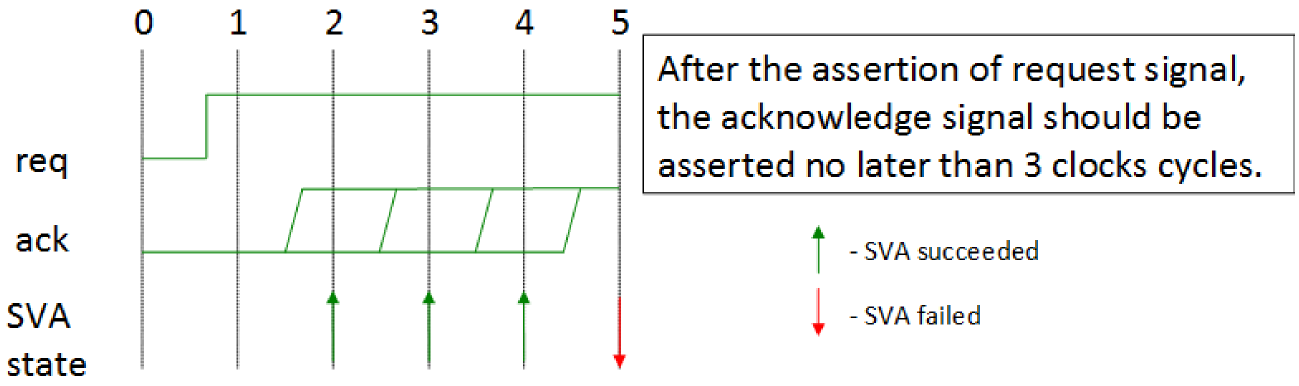
```
ASSERT_LABEL: assert property (req_to_rise_p)
else `uvm_error("ERR", "Assertion failed")
```

Types of SystemVerilog Assertions

- Immediate

```
assert (expression) pass_statement
[else fail_statement]
```

- Concurrent



After the assertion of request signal, the acknowledge signal should be asserted no later than 3 clocks cycles.

Assertions Are Used

- In a verification component
- In a formal proof kit
- In RTL generation

  “Revisiting Regular Expressions in SyntHorus2: from PSL SEREs to Hardware” (Fatemeh (Negin) Javaheri, Katell Morin-Allory, Dominique Borrione)

- For test patterns generation

  “Towards a Toolchain for Assertion-Driven Test Sequence Generation” (Laurence PIERRE)

SVAs Advantages

- Fast
- Non-intrusive
- Flexible
- Coverable

Planning SVA Development

Identify Design Characteristics

- Defined in a document (design specification)

- Known or specified by the designer

- The most common format is of the form *cause and effect:* antecedent |-> consequent

- Antecedent: `$rose(req)`

- Consequent: `##[1:3] $rose(ack)`

Keep it Simple. Partition!

- Complex assertions are typically constructed from complex sequences and properties.

```
a ##1 b[*1:2] |=> c ##1 d[*1:2] |=> $fell(a)
```

⬇

```
sequence seq(arg1, arg2);
 arg1 ##1 arg2[*1:2];
endsequence
```

⬇

```
seq(a, b) |=> seq(c, d) |=> $fell(a)
```

Implementation

Coding Guidelines

- Avoid duplicating design logic in assertions
- Avoid infinite assertions
- Reset considerations
- Mind the sampling clock

Coding Guidelines (contd.)

- Always check for unknown condition (‘X’)
- Assertion naming
- Detailed assertion messages
- Assertion encapsulation

Best Practices

- Review the SVA with the designer to avoid DS misinterpretation
- Use *strong* in assertions that may never complete:

```
assert property ( req |-> strong(##[1:$] ack));
```

- Properties should not hold under certain conditions (reset, enable switch)

```
assert property (
@(posedge clk) disable iff (!setup || !rst_n)
      req |-> strong(##[1:$] ack)
);
```

Best Practices (contd.)

- Avoid overlapping assertions that contradict each other

✓CPU_0:

```
assert property (WRITE |=> ERROR);
```

✓CPU_1:

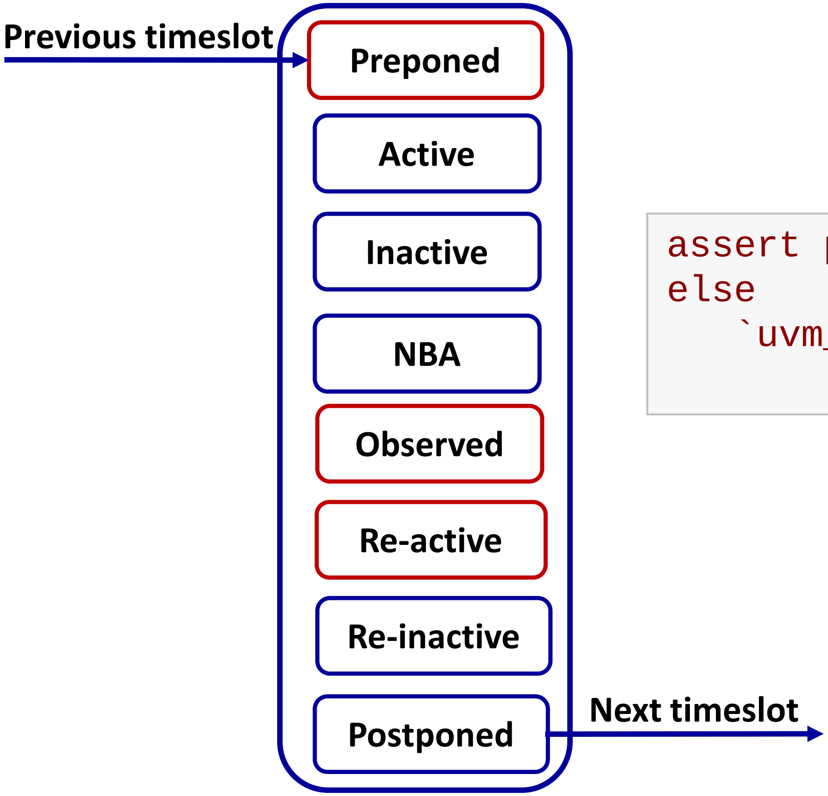
```
assert property (WRITE |=> !ERROR);
```

```
assert property (WRITE and CPU==0 |=> ERROR);
```

```
assert property (WRITE and CPU==1 |=> !ERROR);
```

Best Practices (contd.)

- Use the $sampled() function in action blocks



Previous timeslot

Preponed

Active

Inactive

NBA

Observed

Re-active

Re-inactive

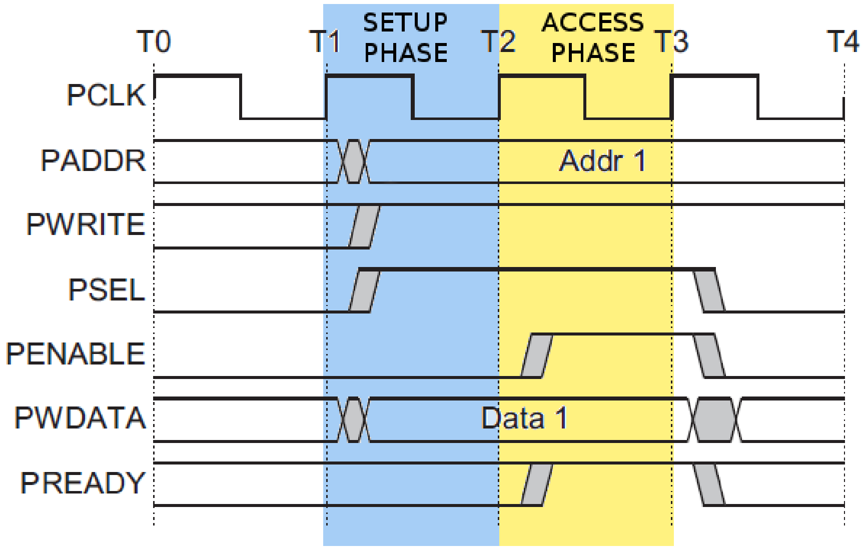
Postponed

Next timeslot

```
assert property ( @(posedge clk) ack == 0 )
else
    `uvm_error("ERROR", $sformatf("Assertion
             failed. ack is %d", $sampled(ack)));
```

Assertion Example

- AMBA APB protocol specification:

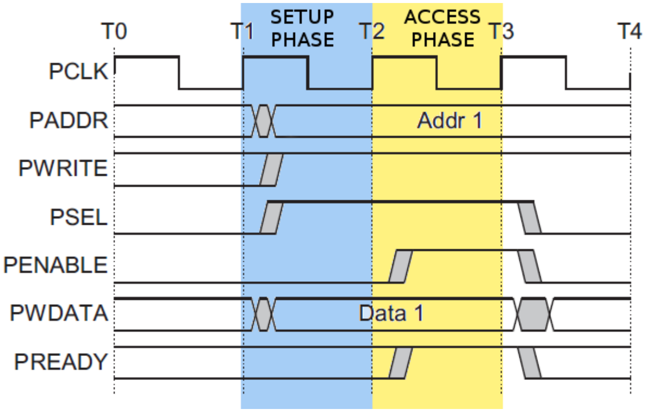


The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.

Assertion Example (contd.)

- Antecedent (the SETUP phase)

```
sequence setup_phase_s;
  $rose(psel) and $rose(pwrite)
  and (!penable) and (!pready);
endsequence
```



- Consequent (the ACCESS phase)

```
sequence access_phase_s;
  $rose(penable)  and $rose(pready) and
  $stable(pwrite) and $stable(pwdata)and
  $stable(paddr)  and $stable(psel);
endsequence
```

Assertion Example (contd.)

- The property can be expressed as:

```
property access_to_setup_p;
  @(posedge clk) disable iff (reset)
  setup_phase_s |=> access_phase_s;
endproperty
```

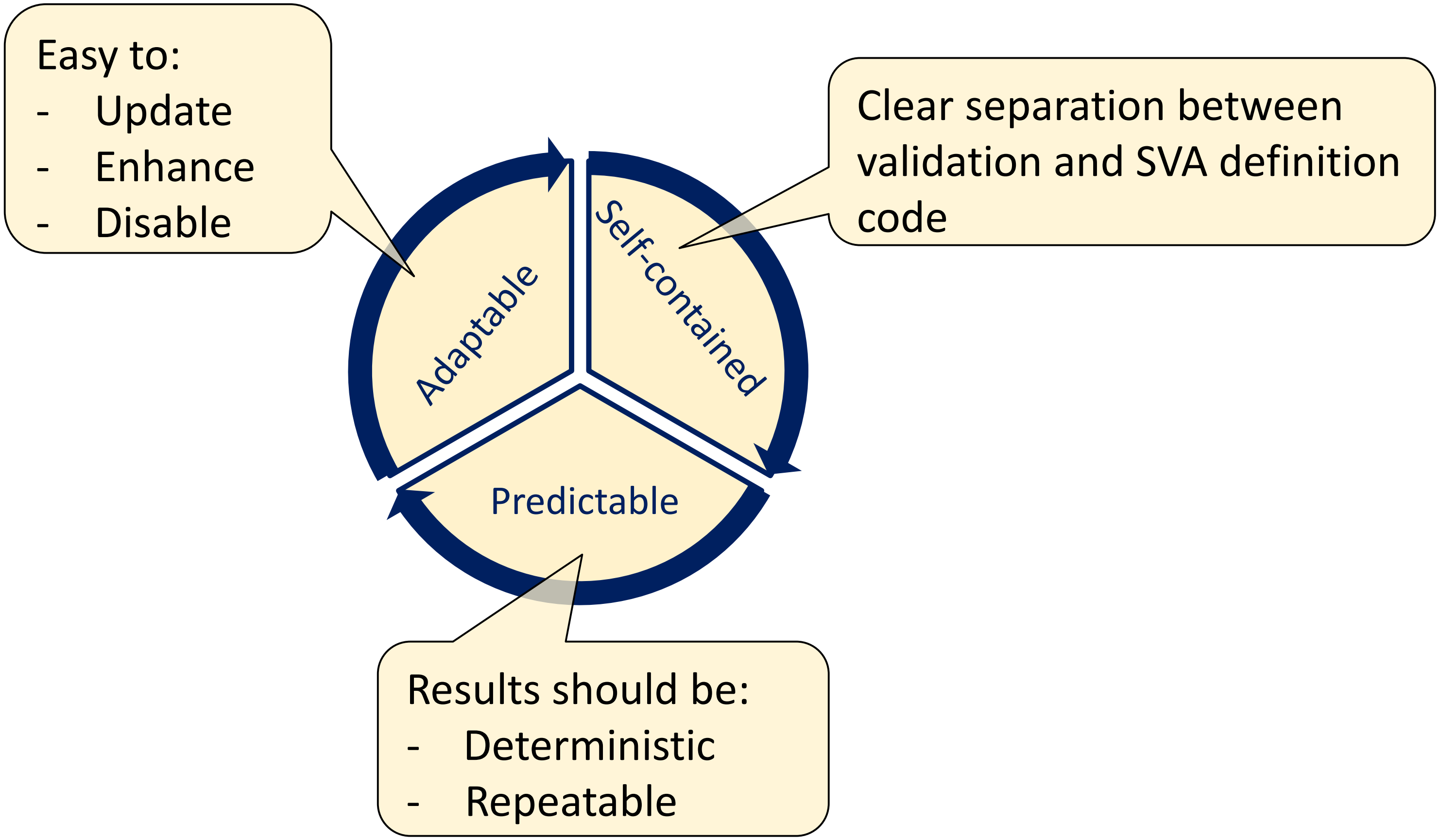
- The assertion will look like:

```
assert property (access_to_setup_p)
else `uvm_error("ERR", "Assertion failed")
```

Does It Work as Intended?

SVA Verification with SVAUnit

SVA Verification Challenges



Easy to:
- Update
- Enhance
- Disable

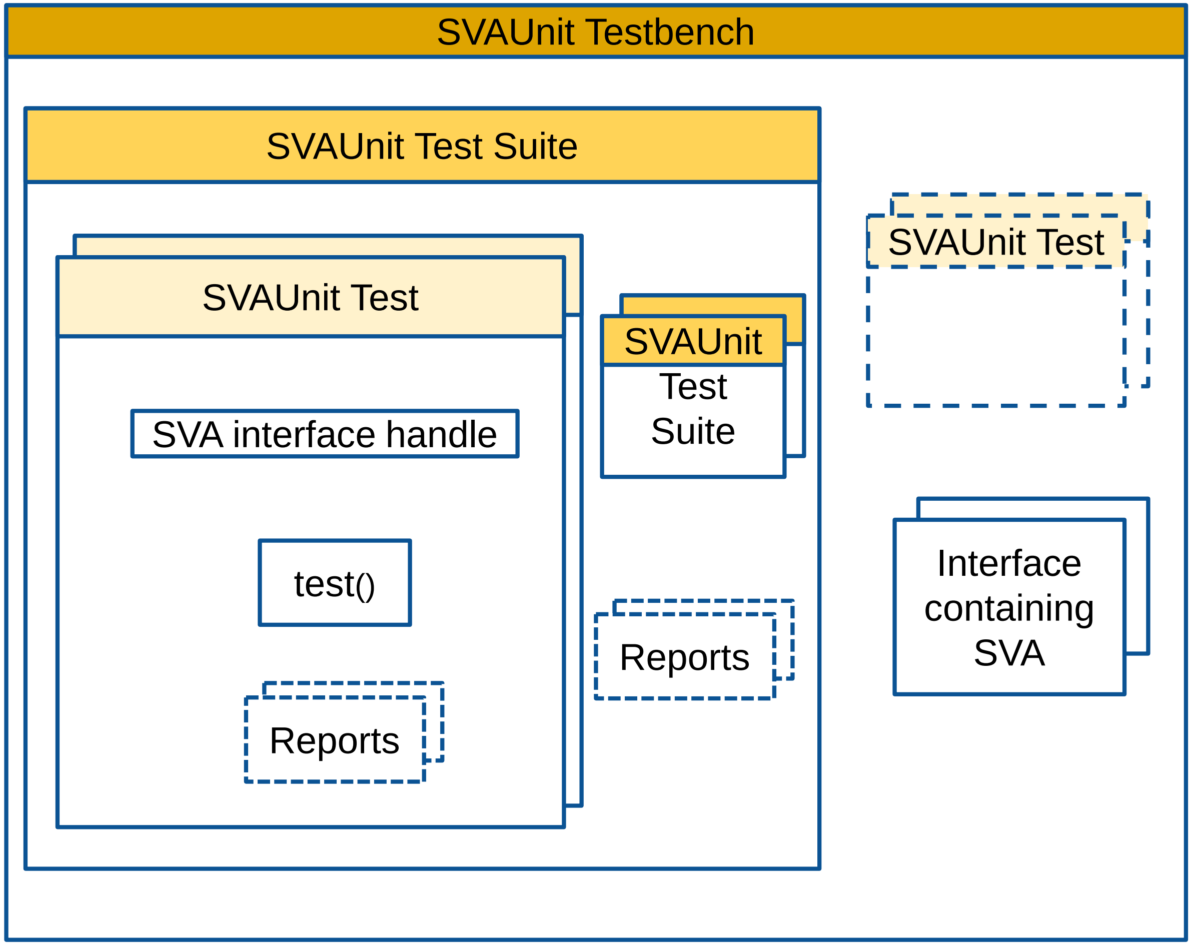
Clear separation between validation and SVA definition code

Results should be:
- Deterministic
- Repeatable

Introducing SVAUnit

- Structured framework for Unit Testing for SVAs
- Allows the user to decouple the SVA definition from its validation code
- UVM compliant package written in SystemVerilog
- Encapsulate each SVA testing scenario inside an unit test
- Easily controlled and supervised using a simple API

SVAUnit Infrastructure



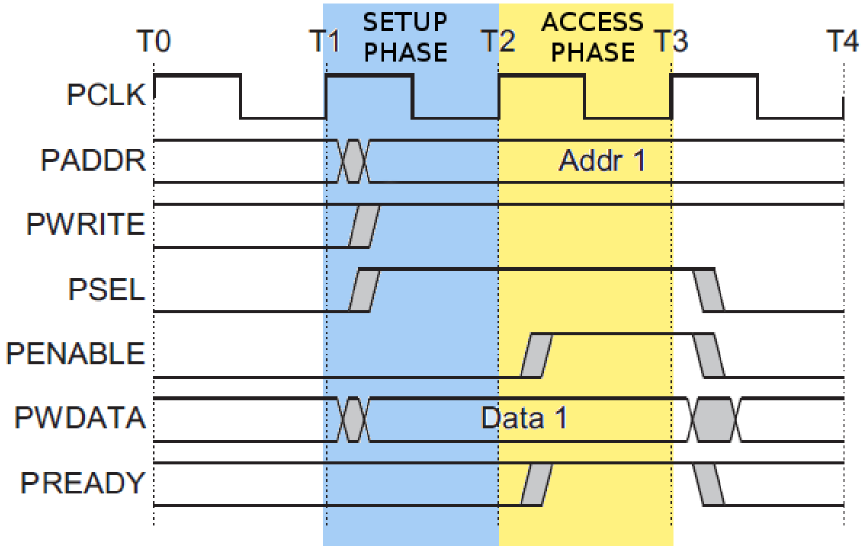
- **SVAUnit Testbench**
  - Enables SVAUnit
  - Instantiates SVA interface
  - Starts test

- **SVAUnit Test**
  - Contains the SVA scenario

- **SVAUnit Test Suite**
  - Test and test suite container

Example Specification

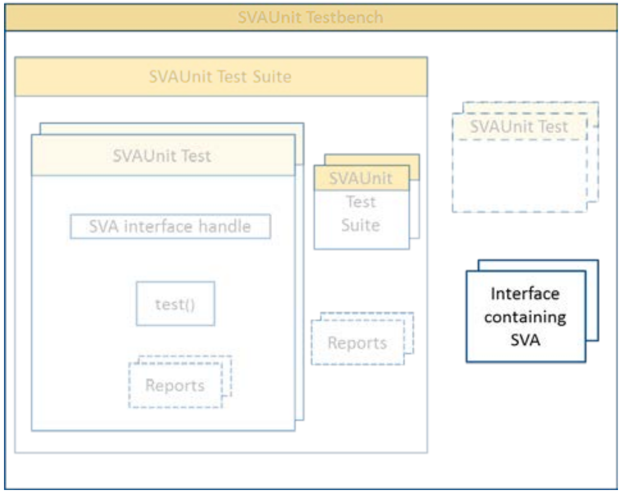
- AMBA APB protocol specification:



The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.

Example APB Interface

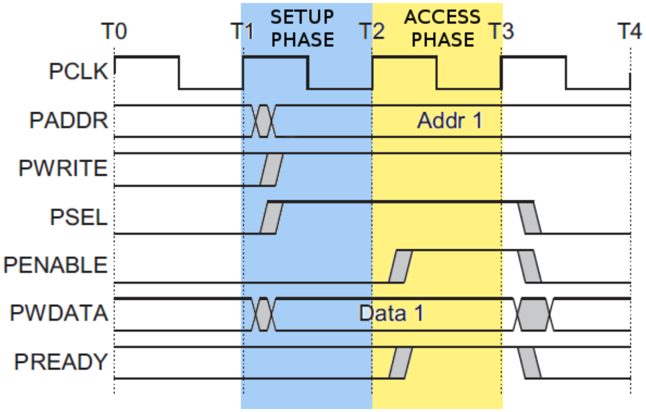
```
interface apb_if (input pclk);
  logic                    psel;
  logic                    pwrite;
  logic                    penable;
  logic                    pready;
  logic [`ADDR_WIDTH-1 :0] paddr;
  logic [`WDATA_WIDTH-1:0] pwdata;

 APB sequences definitions

 APB property definition

 APB assertion definition

endinterface
```



APB Sequences Definitions

- Antecedent (the SETUP phase)

```
sequence setup_phase_s;
  $rose(psel) and $rose(pwrite)
  and (!penable) and (!pready);
endsequence
```



- Consequent (the ACCESS phase)

```
sequence access_phase_s;
  $rose(penable)  and $rose(pready) and
  $stable(pwrite) and $stable(pwdata)and
  $stable(paddr)  and $stable(psel);
endsequence
```

APB Property & Assertion Definitions

- The property can be expressed as:

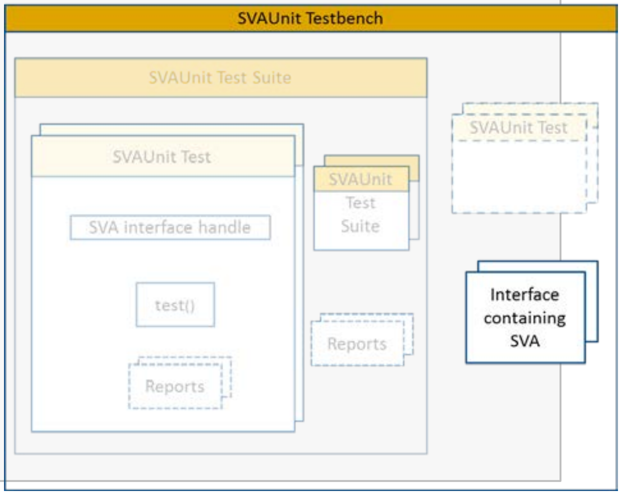
```
property access_to_setup_p;
  @(posedge clk) disable iff (reset)
  setup_phase_s |=> access_phase_s;
endproperty
```

- The assertion will look like:

```
assert property (access_to_setup_p)
else `uvm_error("ERR", "Assertion failed")
```

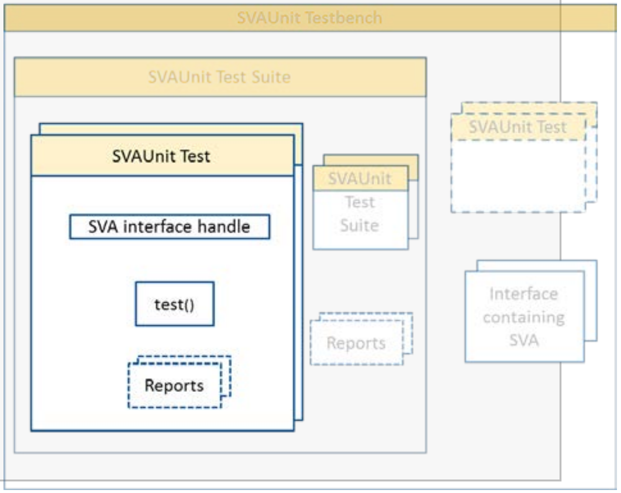
Example of SVAUnit Testbench

```
module top;
  // Instantiate the SVAUnit framework
  `SVAUNIT_UTILS
  ...

  // APB interface with the SVA we want to test
  apb_if an_apb_if(.clk(clock));

  initial begin
    // Register interface with the uvm_config_db
    uvm_config_db#(virtual an_if)::
     set(uvm_root::get(), "*", "VIF", an_apb_if);

     // Start the scenarios
     run_test();
  end

  ...
endmodule
```

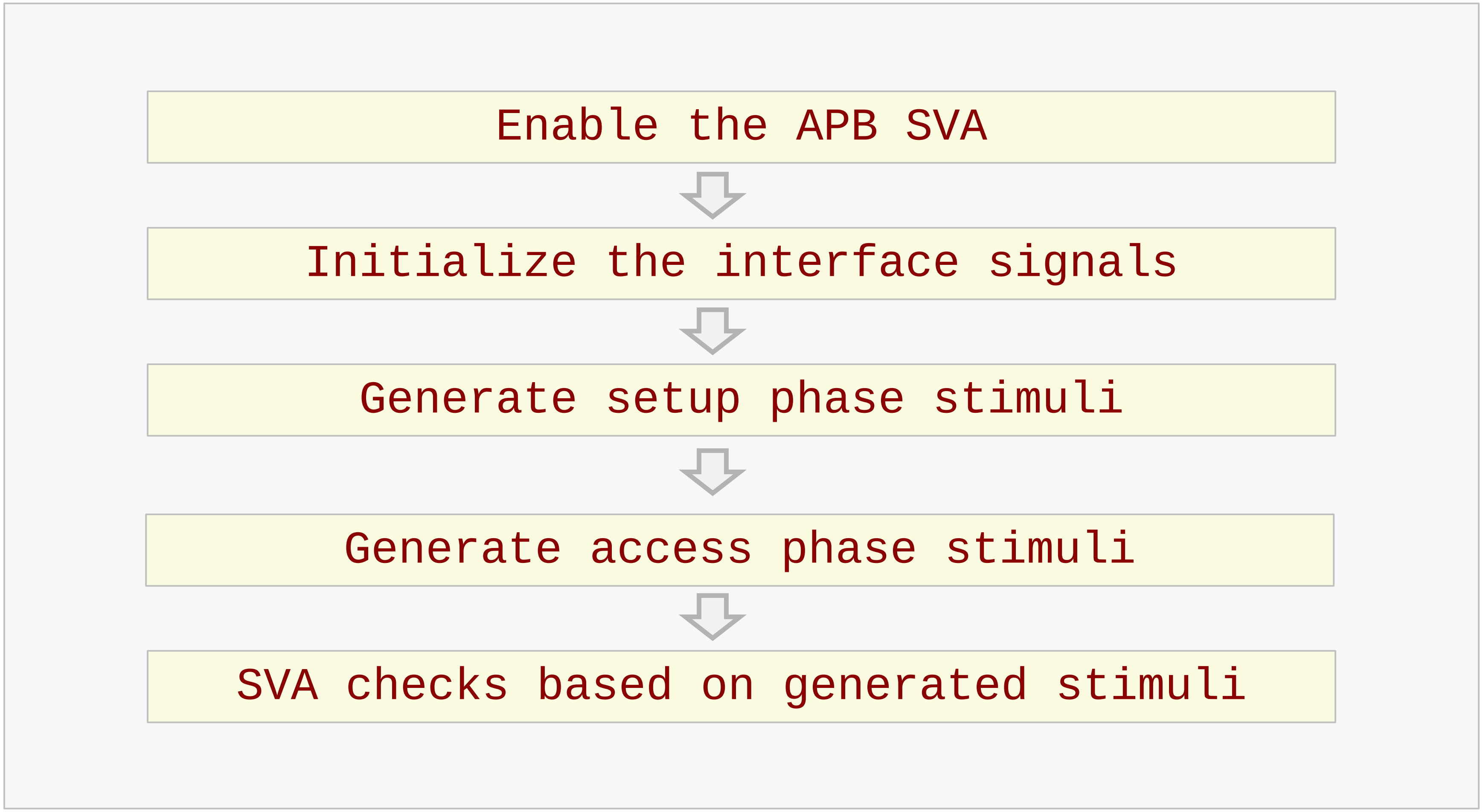


Example of SVAUnit Test

```
class ut1 extends svaunit_test;
   // The virtual interface used to drive the signals
   virtual apb_if apb_vif;

   function void build_phase(input uvm_phase phase);
      // Retrieve the interface handle from the uvm_config_db
      if (!uvm_config_db#(virtual an_if)::get(this, "", "VIF", apb_vif))
        `uvm_fatal("UT1_NO_VIF_ERR", "SVA interface is not set!")

      // Test will run by default;
      disable_test();
   endfunction

   task test();
      // Initialize signals
      // Create scenarios for SVA verification
   endtask
endclass
```



APB – SVAUnit Test Steps



Enable the APB SVA

Initialize the interface signals

Generate setup phase stimuli

Generate access phase stimuli

SVA checks based on generated stimuli

Enable SVA and Initialize Signals

```
...

// Enable the APB SVA
 vpiw.disable_all_assertions();
 vpiw.enable_assertion("APB_PHASES");

 // Initialize signals
 task initialize_signals();
   apb_vif.paddr    <= 32'b0;
   apb_vif.pwdata   <= 32'b0;
   apb_vif.pwrite   <=  1'b0;
   apb_vif.penable  <=  1'b0;
   apb_vif.psel     <=  1'b0;
 endtask

...
```

Generate Setup Phase Stimuli

```
...

  task generate_setup_phase_stimuli(bit valid);
    ...
    // Stimuli for valid SVA scenario
    valid == 1 ->
    pwrite == 1 && psel == 1 && penable == 0 && pready == 0;

    // Stimuli for invalid SVA scenario
    valid == 0 ->
    pwrite != 1 || psel != 1 || penable != 0 || pready != 0;

    ...
  endtask


...
```

Generate Access Phase Stimuli

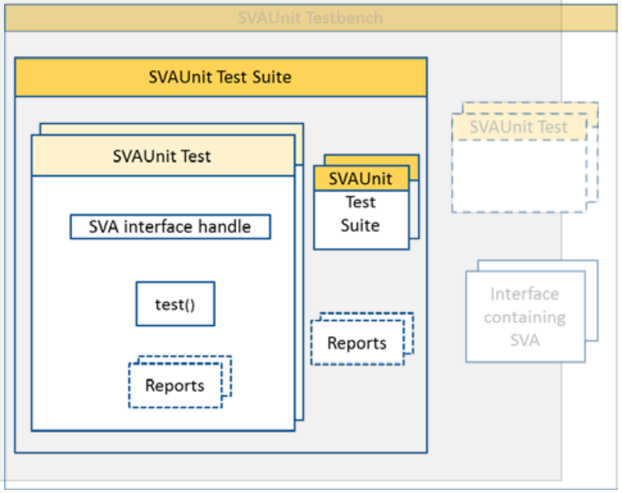
```
...

  task generate_access_phase_stimuli(bit valid);
    ...

     // Constrained stimuli for valid SVA scenario
     valid == 1  ->
     pwdata == apb_vif.pwdata && paddr == apb_vif.paddr &&
     pwrite == 1 && psel == 1 && penable == 1 && pready == 1;


    // Constrained stimuli for invalid SVA scenario
     valid == 0 ->
     pwdata != apb_vif.pwdata ||  paddr != apb_vif.paddr ||
     pwrite != 1 || psel != 1 || penable != 1 || pready != 1;
    ...
  endtask
...
```

SVA State Checking

```
...

 if (valid_setup_phase)
   if (valid_access_phase)
     vpiw.fail_if_sva_not_succeeded("APB_PHASES",
          "The assertion should have succeeded!");
     else
     vpiw.fail_if_sva_succeeded("APB_PHASES",
          "The assertion should have failed!");
 else
  vpiw.pass_if_sva_not_started("APB_PHASES",
       "The assertion should not have started!");

...
```

Example of SVAUnit Test Suite

```
class uts extends svaunit_test_suite;
   // Instantiate the SVAUnit tests
   ut1 ut1;
   ...
   ut10 ut10;

   function void build_phase(input uvm_phase phase);
     // Create the tests using UVM factory
     ut1 = ut1::type_id::create("ut1", this);
     ...
     ut10 = ut10::type_id::create("ut10", this);

     // Register tests in suite
     `add_test(ut1);
     ...
     `add_test(ut10);
   endfunction

endclass
```



SVAUnit Test API

CONTROL

- disable_all_assertions();
- enable_assertion(sva_name);
- enable_all_assertions();

. . .

CHECK

- fail_if_sva_does_not_exists(sva_name, error_msg);
- pass_if_sva_not_succeeded(sva_name, error_msg);
- pass/fail_if(expression, error_msg);

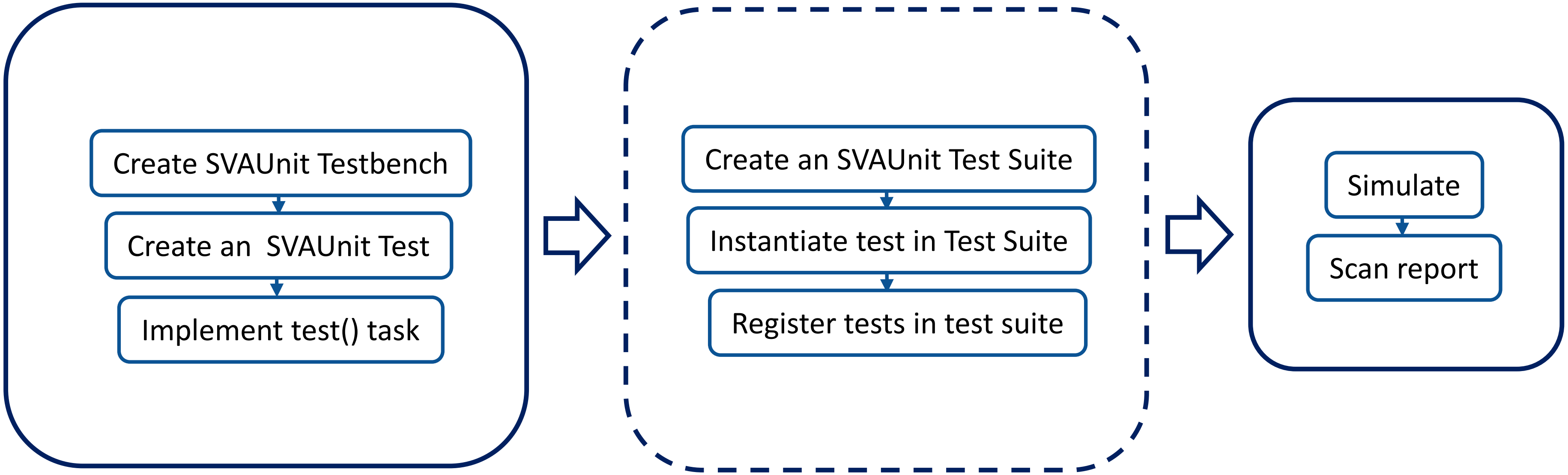
. . .

REPORT

- print_status();
- print_sva();
- print_report();

. . .

SVAUnit Flow



Create SVAUnit Testbench

Create an SVAUnit Test

Implement test() task

Create an SVAUnit Test Suite

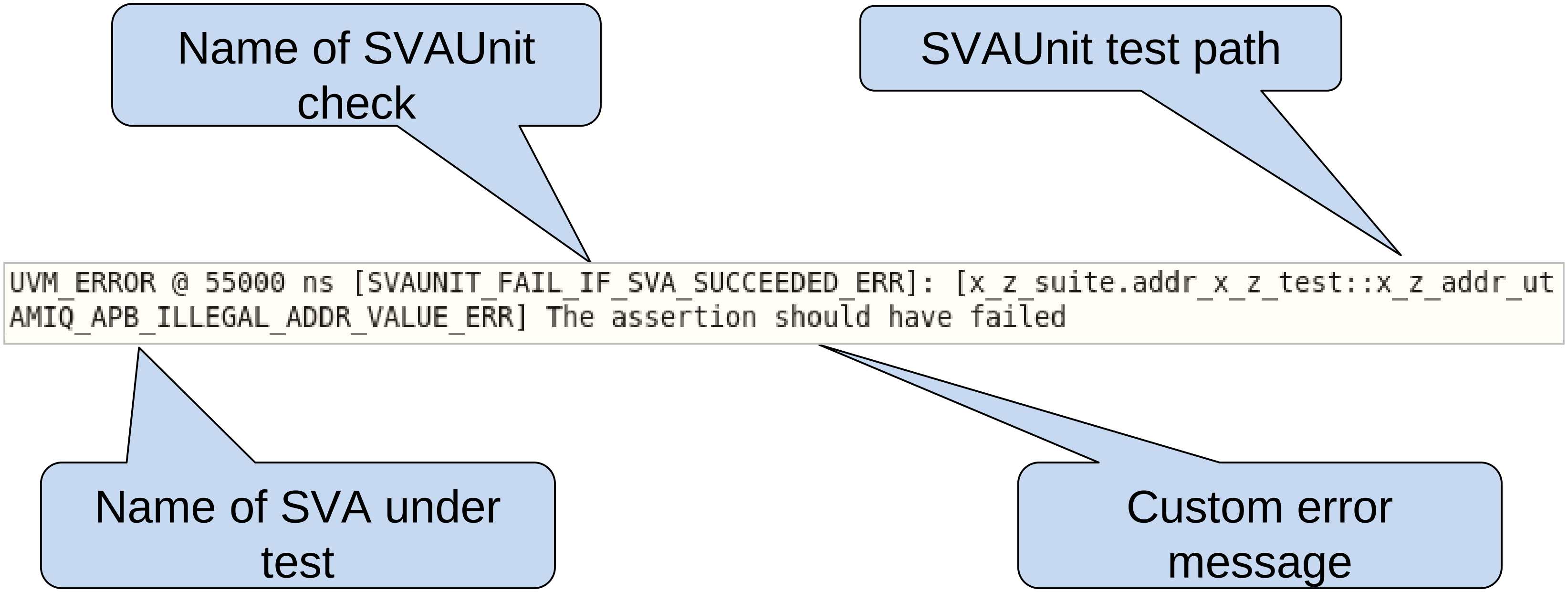
Instantiate test in Test Suite

Register tests in test suite

Simulate

Scan report

Error Reporting



```
UVM_ERROR @ 55000 ns [SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR]: [x_z_suite.addr_x_z_test::x_z_addr_ut
AMIQ_APB_ILLEGAL_ADDR_VALUE_ERR] The assertion should have failed
```

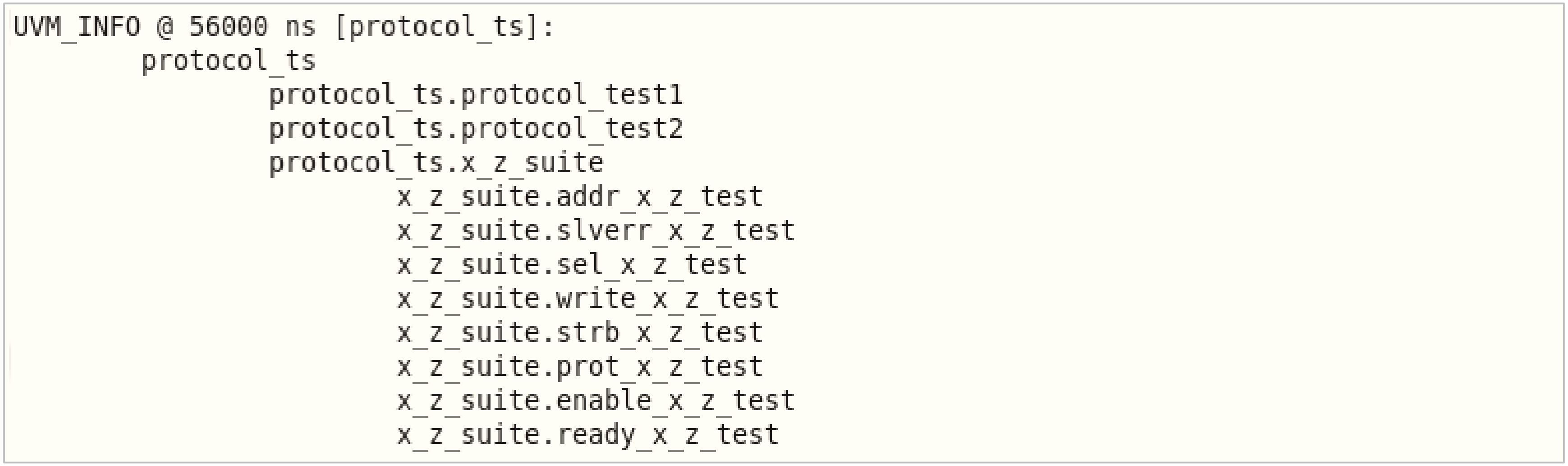
Name of SVAUnit check

SVAUnit test path

Name of SVA under test

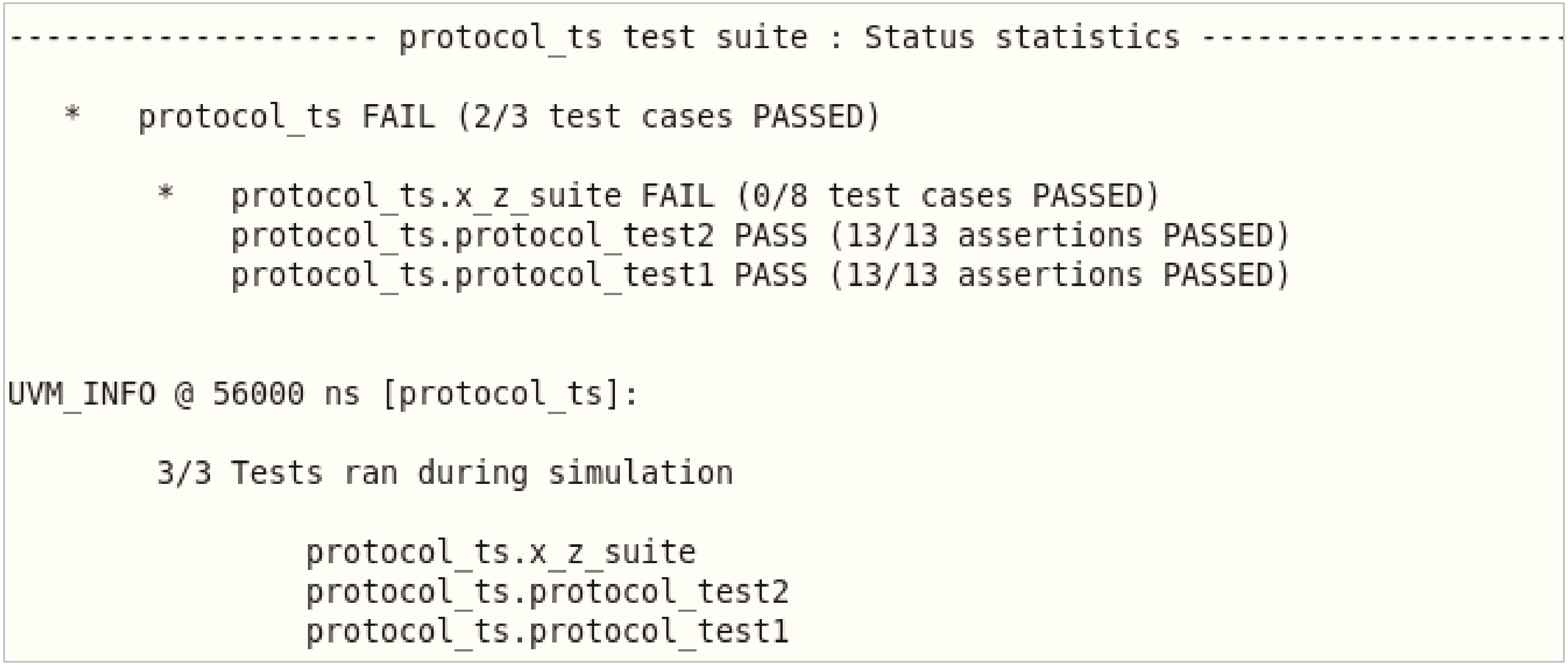
Custom error message

Hierarchy Report



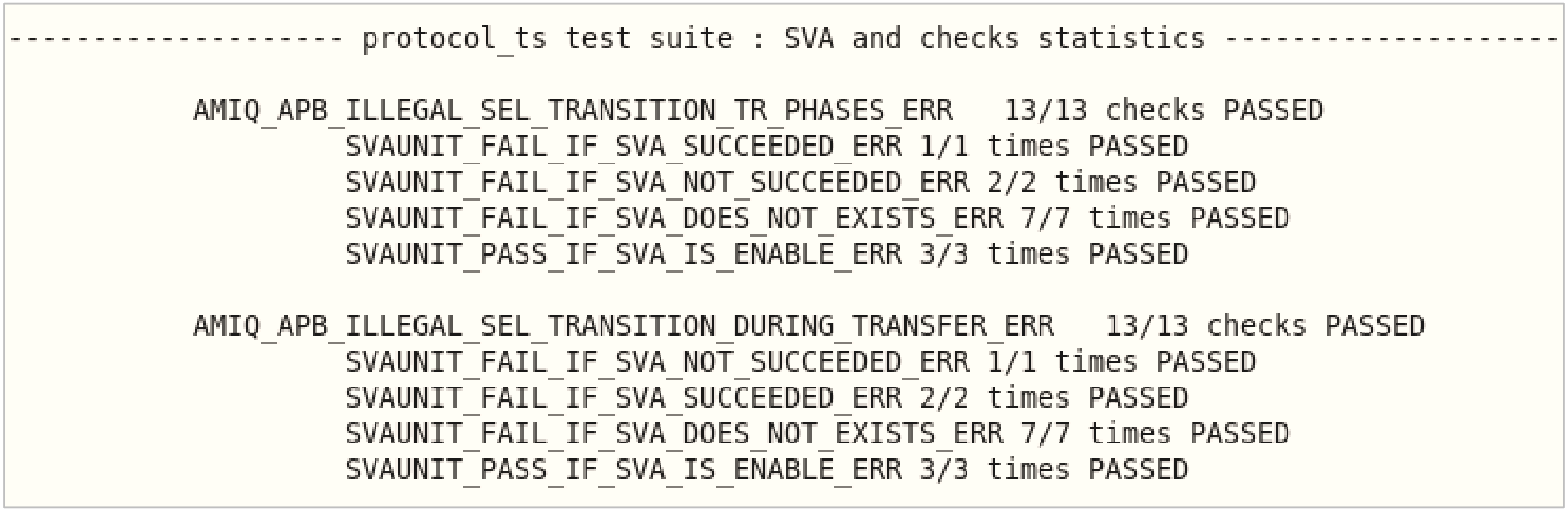
```
UVM_INFO @ 56000 ns [protocol_ts]:
        protocol_ts
                protocol_ts.protocol_test1
                protocol_ts.protocol_test2
                protocol_ts.x_z_suite
                        x_z_suite.addr_x_z_test
                        x_z_suite.slverr_x_z_test
                        x_z_suite.sel_x_z_test
                        x_z_suite.write_x_z_test
                        x_z_suite.strb_x_z_test
                        x_z_suite.prot_x_z_test
                        x_z_suite.enable_x_z_test
                        x_z_suite.ready_x_z_test
```

Test Scenarios Exercised



```
-------------------- protocol_ts test suite : Status statistics --------------------

   *   protocol_ts FAIL (2/3 test cases PASSED)

        *   protocol_ts.x_z_suite FAIL (0/8 test cases PASSED)
            protocol_ts.protocol_test2 PASS (13/13 assertions PASSED)
            protocol_ts.protocol_test1 PASS (13/13 assertions PASSED)


UVM_INFO @ 56000 ns [protocol_ts]:

        3/3 Tests ran during simulation

                protocol_ts.x_z_suite
                protocol_ts.protocol_test2
                protocol_ts.protocol_test1
```

SVAs and Checks Exercised



```
-------------------- protocol_ts test suite : SVA and checks statistics --------------------

           AMIQ_APB_ILLEGAL_SEL_TRANSITION_TR_PHASES_ERR   13/13 checks PASSED
                    SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 1/1 times PASSED
                    SVAUNIT_FAIL_IF_SVA_NOT_SUCCEEDED_ERR 2/2 times PASSED
                    SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 7/7 times PASSED
                    SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 3/3 times PASSED

           AMIQ_APB_ILLEGAL_SEL_TRANSITION_DURING_TRANSFER_ERR   13/13 checks PASSED
                    SVAUNIT_FAIL_IF_SVA_NOT_SUCCEEDED_ERR 1/1 times PASSED
                    SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 2/2 times PASSED
                    SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 7/7 times PASSED
                    SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 3/3 times PASSED
```

SVA Test Patterns

Simple Implication Test

- a and b |=> c

```
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_b, stimuli_for_c);

    interface.a  <= stimuli_for_a;
    interface.b  <= stimuli_for_b;
    @(posedge an_vif.clk);

    interface.c  <= stimuli_for_c;
    @(posedge interface.clk);

    @(posedge interface.clk);
    if (stimuli_for_a == 1 && stimuli_for_b == 1)
      if (stimuli_for_c == 1)
        vpiw.fail_if_sva_not_succeeded("IMPLICATION_ASSERT",
             "The assertion should have succeeded!");
        else
          vpiw.fail_if_sva_succeeded("IMPLICATION_ASSERT",
             "The assertion should have failed!");
     else
        vpiw.pass_if_sva_not_started("IMPLICATION_ASSERT",
             "The assertion should not have started!");
end
```

Multi-thread Antecedent/Consequent

- `$rose(a) ##[1:4] b |-> ##[1:3] c`

```
repeat (test_loop_count) begin
    // Generate valid delays for asserting b and c signals
    randomize(delay_for_b inside {[1:4]}, delay_for_c inside {[1:3]});
    interface.a  <= 1;

    repeat (delay_for_b)
      @(posedge interface.clk);
    interface.b  <= 1;

    vpiw.pass_if_sva_started_but_not_finished("MULTITHREAD_ASSERT",
             "The assertion should have started but not finished!");

    repeat (delay_for_c)
      @(posedge interface.clk);
    interface.c  <= 1;

    vpiw.pass_if_sva_succeeded("MULTITHREAD_ASSERT",
             "The assertion should have succeeded!");

end
```

Multi-thread Antecedent/Consequent (contd.)

- `$rose(a) ##[1:4] b |-> ##[1:3] c`

```
repeat (test_loop_count) begin
    // Generate invalid delays for asserting b and c signals
    randomize(delay_for_b inside {[0:10]}, delay_for_c inside {0,[4:10]});
    interface.a  <= 1;

    repeat (delay_for_b)
      @(posedge interface.clk);
    interface.b  <= 1;

    vpiw.pass_if_sva_not_succeeded("MULTITHREAD_ASSERT",
             "The assertion should have failed!");

    repeat (delay_for_c)
      @(posedge interface.clk);
    interface.c  <= 1;

    if (delay_for_b < 5)
    vpiw.fail_if_sva_succeeded("MULTITHREAD_ASSERT",
             "The assertion should have failed!");
end
```

Consecutive Repetition

- a |-> b[*1:2] ##1 c

```
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_c, number_of_b_cycles <= 2);

    interface.a  <= stimuli_for_a;

    repeat (number_of_b_cycles) begin
      randomize(stimuli_for_b)
      interface.b  <= stimuli_for_b;
      if (stimuli_for_b == 1) number_of_b_assertions += 1;

      @(posedge interface.clk);
    end

    if (stimuli_for_a == 1 && number_of_b_assertions == number_of_b_cycles &&
        number_of_b_assertions > 0)
      vpiw.pass_if_sva_started_but_not_finished("IMPLICATION_ASSERT",
           "The assertion should have started but not finished!");
    @(posedge interface.clk);

... // (continued on the next slide)
```

Consecutive Repetition (contd.)

- a |-> b[*1:2] ##1 c

```
...
// (continued from previous slide)

    interface.c  <= stimuli_for_c;
    @(posedge interface.clk);

    if (stimuli_for_a == 1)
        if (number_of_b_assertions != number_of_b_cycles ||
            number_of_b_assertions == 0 ||
            stimuli_for_c == 0)
          vpiw.fail_if_sva_succeeded("IMPLICATION_ASSERT",
               "The assertion should have failed!");
        else
          vpiw.fail_if_sva_not_succeeded("IMPLICATION_ASSERT",
              "The assertion should have succeeded!");

end // end of test repeat loop
```

Repetition Range with Zero

- a |-> b[*0:2] ##1 c

```
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_c, number_of_b_cycles <= 2);

    interface.a  <= stimuli_for_a;

    repeat (number_of_b_cycles) begin
      randomize(stimuli_for_b)
      interface.b  <= stimuli_for_b;
      if (stimuli_for_b == 1) number_of_b_assertions += 1;

      @(posedge interface.clk);
    end

    if (stimuli_for_a == 1 && number_of_b_assertions == number_of_b_cycles)
        ~~&& number_of_b_assertions > 0)~~
      vpiw.pass_if_sva_started_but_not_finished("IMPLICATION_ASSERT",
           "The assertion should have started but not finished!");
    @(posedge interface.clk);

... // (continued on the next slide)
```

Repetition Range with Zero (contd.)

- a |-> b[*0:2] ##1 c

```
...
// (continued from previous slide)

    interface.c  <= stimuli_for_c;
    @(posedge interface.clk);

    if (stimuli_for_a == 1)
        if (number_of_b_assertions != number_of_b_cycles ||
            ~~number_of_b_assertions == 0 ||~~
            stimuli_for_c == 0)
          vpiw.fail_if_sva_succeeded("REPETITION_RANGE0_ASSERT",
               "The assertion should have failed!");
        else
          vpiw.fail_if_sva_not_succeeded("REPETITION_RANGE0_ASSERT",
              "The assertion should have succeeded!");

end // end of test repeat loop
```

Sequence Disjunction

- `a |=> (b ##1 c) or (d ##1 e)`

```
repeat (test_loop_count) begin
  randomize(stimuli_for_a, stimuli_for_b, stimuli_for_c, stimuli_for_d, stimuli_for_e);

  interface.a  <= stimuli_for_a;
  @(posedge interface.clk);
  fork
    begin
              Stimuli for branch: (b ##1 c)
         SVA state check based on branch stimuli
    end
    begin
              Stimuli for branch: (d ##1 e)
         SVA state check based on branch stimuli
    end
    join
end
```

Sequence Disjunction (contd.)

- a |=> (b ##1 c) or (d ##1 e)

```
...

  // Stimuli for branch (b ##1 c)
  fork
    begin
      interface.b  <= stimuli_for_b;
      @(posedge interface.clk);

      interface.c  <= stimuli_for_c;
      @(posedge interface.clk);

      @(posedge interface.clk);
      // SVA state check based on branch stimuli
      sva_check_phase(interface.a, interface.b, interface.c);
     end
   join
```

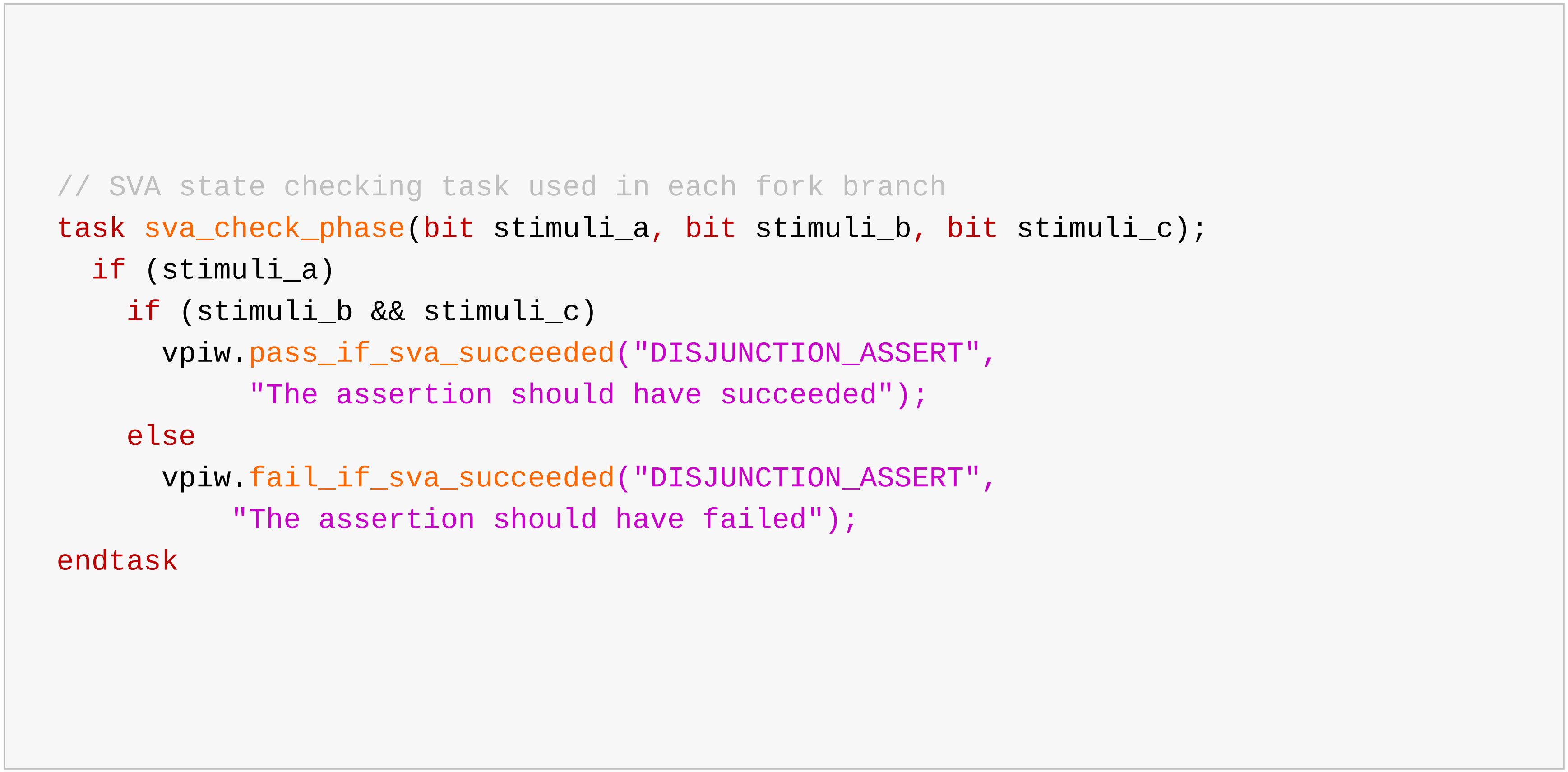
Sequence Disjunction (contd.)

- `a |=> (b ##1 c) or (d ##1 e)`

```
...

  // Stimuli for branch (d ##1 e)
  fork
    begin
      interface.b  <= stimuli_for_d;
      @(posedge interface.clk);

      interface.c  <= stimuli_for_e;
      @(posedge interface.clk);

      @(posedge interface.clk);
      // SVA state check based on branch stimuli
      sva_check_phase(interface.a, interface.d, interface.e);
     end
   join
```

Sequence Disjunction (contd.)

- a |=> (b ##1 c) or (d ##1 e)

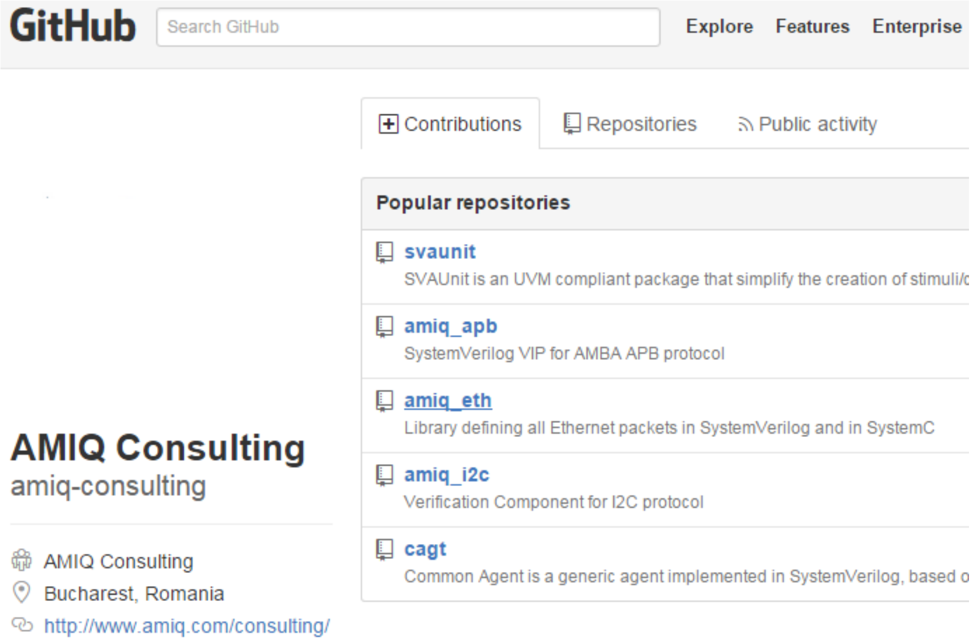


```
// SVA state checking task used in each fork branch
task sva_check_phase(bit stimuli_a, bit stimuli_b, bit stimuli_c);
  if (stimuli_a)
    if (stimuli_b && stimuli_c)
      vpiw.pass_if_sva_succeeded("DISJUNCTION_ASSERT",
           "The assertion should have succeeded");
    else
      vpiw.fail_if_sva_succeeded("DISJUNCTION_ASSERT",
          "The assertion should have failed");
endtask
```

Tools Integration

Simulator independent!

Availability



GitHub Search GitHub Explore Features Enterprise

Contributions Repositories Public activity

AMIQ Consulting
amiq-consulting

AMIQ Consulting
Bucharest, Romania
http://www.amiq.com/consulting/

Popular repositories

- **svaunit** – SVAUnit is an UVM compliant package that simplify the creation of stimuli/c
- **amiq_apb** – SystemVerilog VIP for AMBA APB protocol
- **amiq_eth** – Library defining all Ethernet packets in SystemVerilog and in SystemC
- **amiq_i2c** – Verification Component for I2C protocol
- **cagt** – Common Agent is a generic agent implemented in SystemVerilog, based o

- SVAUnit is an open-source package released by AMIQ Consulting

- We provide:

- SystemVerilog and simulator integration codes
- AMBA-APB assertion package
- Code templates and examples
- HTML documentation for API

https://github.com/amiq-consulting/svaunit

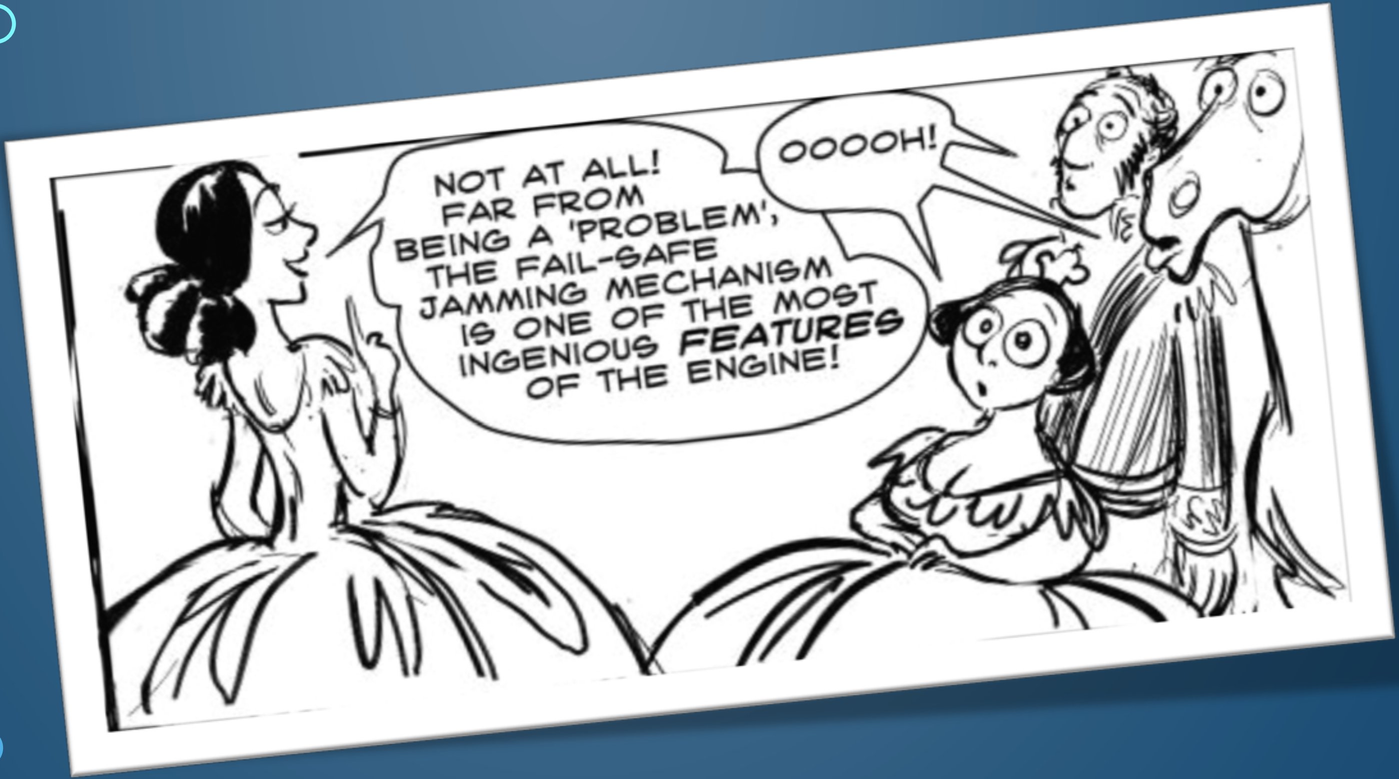
Conclusions

- SVAUnit decouples the checking logic from SVA definition code
- Safety net for eventual code refactoring
- Can also be used as self-checking documentation on how SVAs work
- Quick learning curve
- Easy-to-use and flexible API
- Speed up verification closure
- Boost verification quality

Thank you!

FORMAL SPECIFICATION, SYSTEM VERILOG ASSERTIONS & COVERAGE

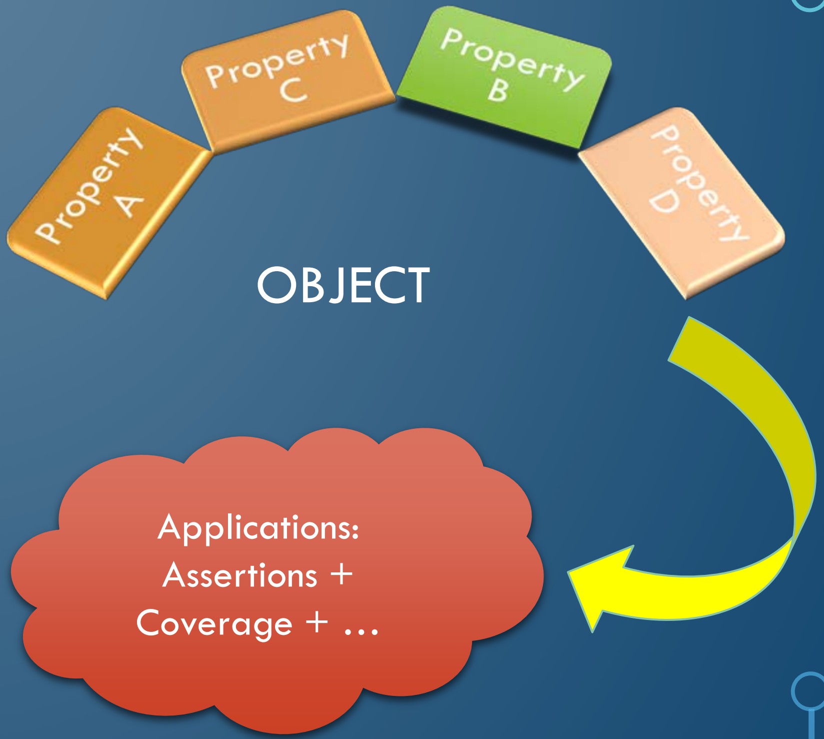
By Calderón-Rico, Rodrigo & Tapia Sanchez, Israel G.


NOT AT ALL! FAR FROM BEING A 'PROBLEM', THE FAIL-SAFE JAMMING MECHANISM IS ONE OF THE MOST INGENIOUS **FEATURES** OF THE ENGINE!
OOOOH!

OBJECTIVE

- ❑ **Learn how to define objects by specifying their properties which are formally described.**

- ❑ **Using the formal specification for assertion or coverage purposes with real examples and gain comparisons versus other methods as scripting and SystemVerilog always blocks.**



AGENDA

I. Introduction

- ❑ Why do we need formal specification?
- ❑ Formal Specification Components
- ❑ Layers of Assertion Language
- ❑ Temporal Logic

II. Language constructs

- ❑ Definition
- ❑ Boolean logic connectors
- ❑ Temporal logic connectors
- ❑ Sequence Specification
- ❑ Sequence Declaration
- ❑ Property Declaration

III. Sequence

- ❑ Basics
- ❑ Sequence construction.
  - o Temporal logic connectors
  - o Additional sequence features
- ❑ USB Examples

IV. Property

- ❑ Basics
- ❑ Property construction
- ❑ Examples

V. Assertion Language

- ❑ Introduction
- ❑ Assert Language

VI. Comparative Results

I - INTRODUCTION

WHY DO WE NEED FORMAL SPECIFICATION?

- Formal specification languages are used to describe design properties unambiguously and precisely.
- Usually properties are written as part of the high level design specifications in a text document. But writing specification in a natural language is ambiguous.
- Consider the following typical property specification: Each request should be granted in four clock cycles. This specification is ambiguous:
  - Do we count four clock cycles starting from the cycle when the request was issued, or from the next cycle?
  - Do we require that the grant is issued during the first four cycles or exactly at the fourth cycle?
  - May two requests be served by the same grant or should they be served by two separate grants?

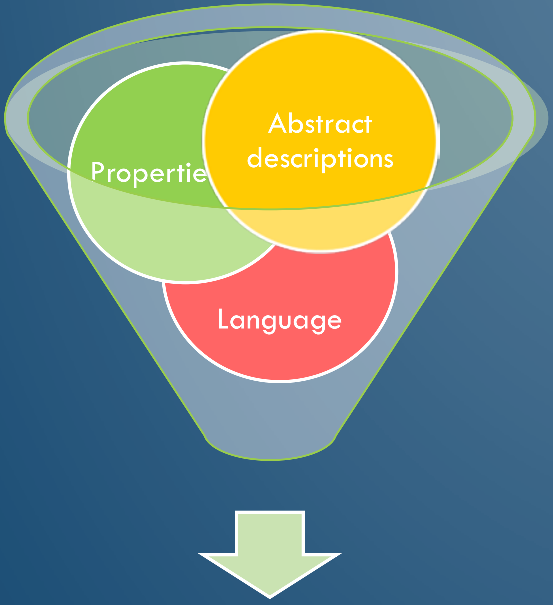
❑ The same specification written in SystemVerilog Asserions(SVA) is unambiguous:

```
assert property( @( posedge clk )  request |-> ##4 grant );
```

❑ This specification defines a clocked, or concurrent assertion, and it reads: when request is issued, it should be followed by grant in the fourth clock cycle measured from the clock cycle when request was issued.

❑ Because of the formal nature of SVA, specifications can be interpreted by tools, and what is more important, understood by humans. When the specifications are formally defined, there is no place for misunderstanding.

FORMAL SPECIFICATION COMPONENTS



Formal Specification

❑Abstract descriptions are aimed to specify an abstract behavior as it defines **what** happens and **when**, without specifying **how** exactly happened.

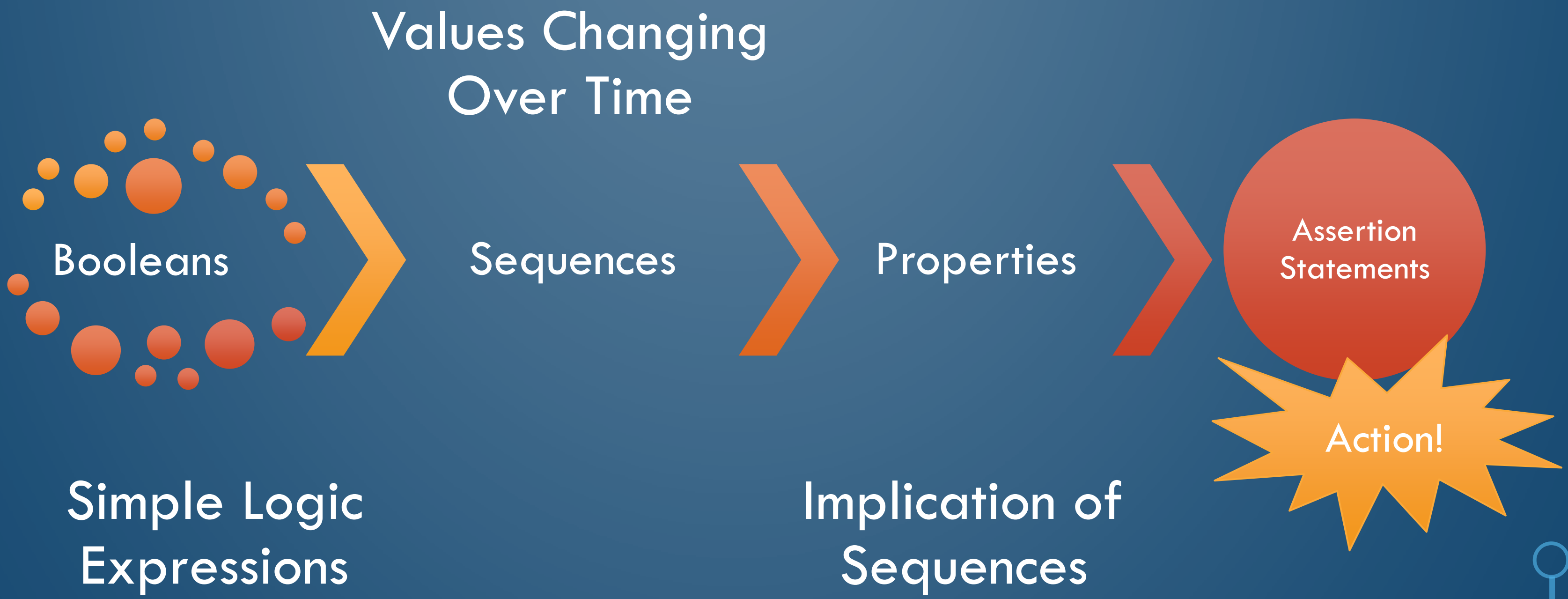
❑Abstract descriptions are encapsulated in properties.

❑A group of properties may describe a complete model.

❑Application:

Pre-si verification: The model created via formal properties is a way of creating evidence suggesting that a system either does or does not have some behavior.

LAYERS OF SVA ASSERTION LANGUAGE



Values Changing Over Time

Booleans

Sequences

Properties

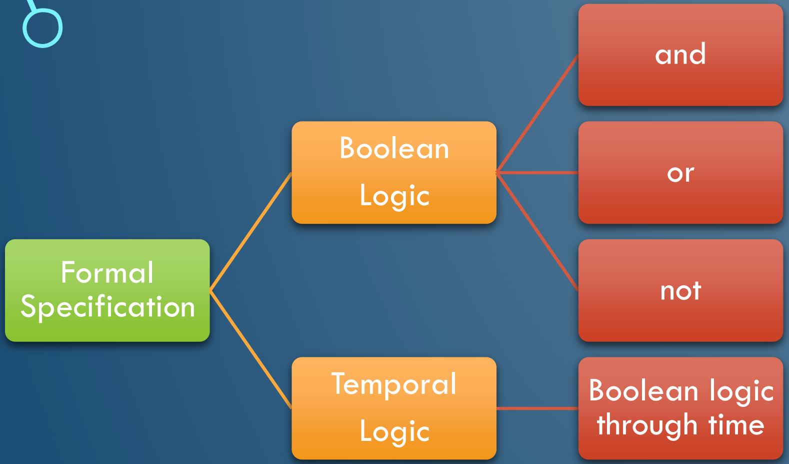
Assertion Statements

Action!

Simple Logic Expressions

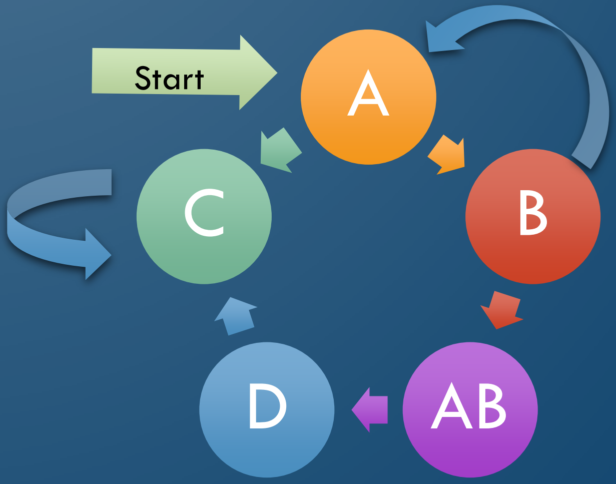
Implication of Sequences

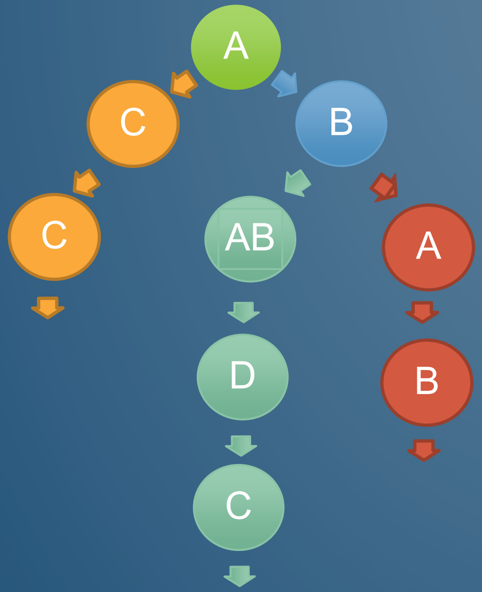
TEMPORAL LOGIC



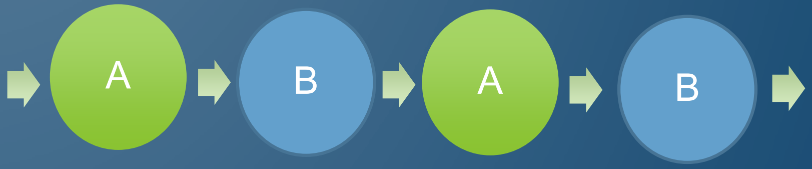
- ❑ One can associate temporal logic to a path through the time where a sequence of events occur in a specified order. The events are constructed via Boolean logic.
- ❑ Kripke structures (nondeterministic finite state machines) set a model to evaluate temporal logic in discrete time.

Note: Any temporal logic statement is assumed to be in the context of discrete time, and may or may not be specified in a discrete event context.





A single tree branch can be understood as a realization of one possible path through time.



The tree structure can help to unfold a state diagram in order to separate possible different paths.

II - LANGUAGE CONSTRUCTS

Definition

❑ The chosen language is SystemVerilog (SV).

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

- Abstracts a detailed specification of the design.
- Specification of assertions coverage and testbench verification that is based on manual or automatic methodologies.

The syntax defined in SV is to generate abstract descriptions (properties).

Boolean logic connectors

- **and**
- **or**
- Non-temporal implication:

  expression 1 **|->** expression 2 (if 1 then 2)

❑ Distribution:

```
expression dist { dist_list } ;

dist_list      :=   dist_item { , dist_item }
dist_item      :=   value_range [ dist_weight ]
dist_weight  :=  (:= expression) |  (:/ expression)
```

The distribution operator **dist** evaluates to true if the value of the expression is contained in the set; otherwise, it evaluates to false.

Example:

```
usb_symbol dist {100 := 1, 200 := 2, 300 := 5}
```

It means **usb_symbol** is equal to 100, 200, or 300 with weighted ratio of 1-2-5.

TEMPORAL LOGIC CONNECTORS

❑ Delay range : ##

## integral_number or identifier

## (constant_expression)

## [cycle_delay_const_range_expression]

❑ Temporal implication: expression 1 |=> expression 2

❑ Consecutive repetition: [* const_or_range_expression]

❑ Non-consecutive repetition: [= const_or_range_expression]

❑ Go-to repetition: [- > const_or_range_expression]

Sequence specification (temporal sequence)

- □ throughout
- □ within
- □ first_match
- □ intersect

Sequence declaration:

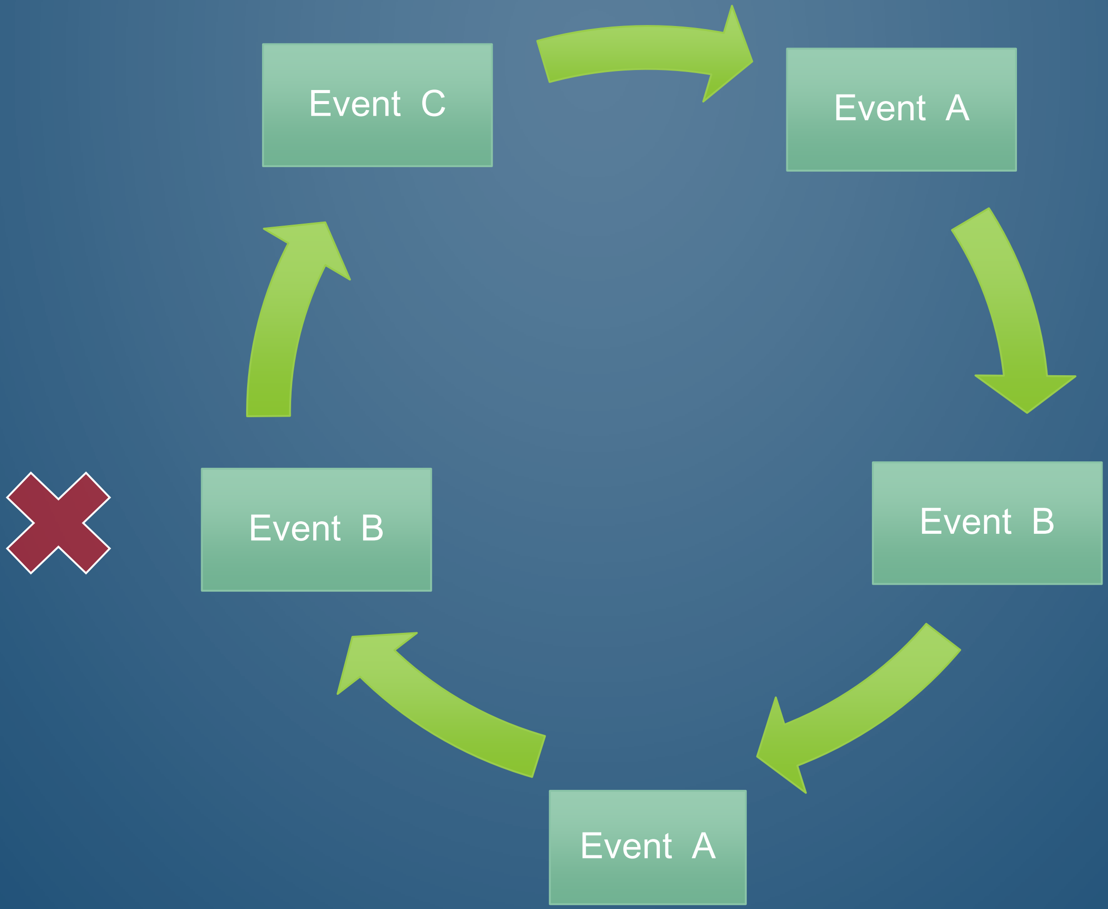
- □ **sequence** name [(sequence_variables)] **endsequence**
- □ Encapsulates a temporal proposition to make it reusable.

Property declaration:

- □ **property** name [(property_variables)] **endproperty**
- □ Encapsulates an abstract description to make it reusable.

III - SEQUENCE

BASICS



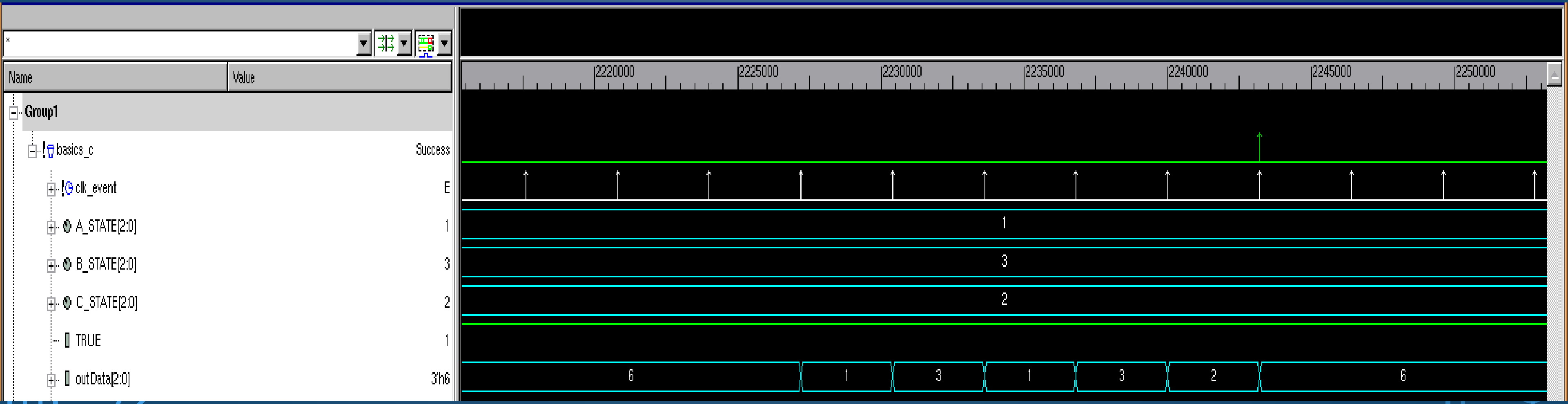
Properties are very often constructed out of sequential behaviors, thus, the sequence feature provides the capability to build and manipulate sequential behaviors.

In SV a sequence can be declared in:

I. a module,
II. an interface,
III. a program,
IV. a clocking block,
V. a package
VI. a compilation-unit scope

Example:

```
sequence basics_c;
@( posedge clk ) A_STATE ##1 B_STATE ##1 A_STATE ##1 B_STATE ##1 C_STATE;
endsequence
```



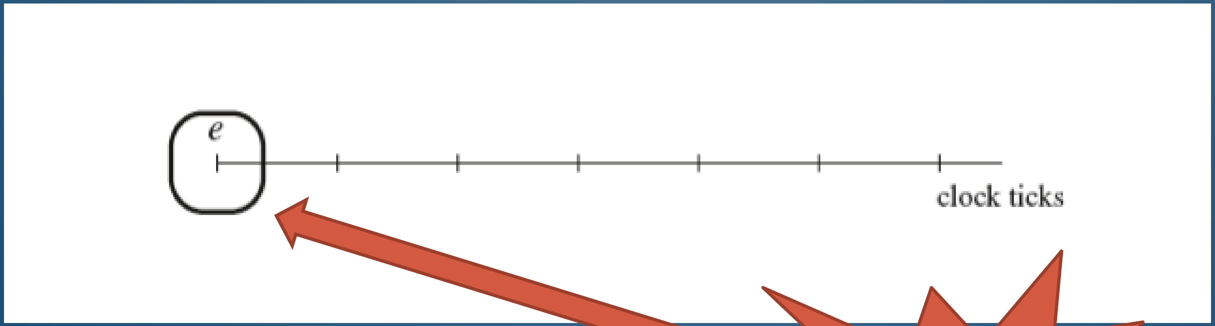
Name	Value
Group1	
basics_c	Success
clk_event	E
A_STATE[2:0]	1
B_STATE[2:0]	3
C_STATE[2:0]	2
TRUE	1
outData[2:0]	3'h6

Sequence Construction

Boolean expression **e** defines the simplest sequence – a **Boolean sequence**

❑ This sequence has a match at its initial point if **e** is true

❑ Otherwise, it does not have any satisfaction points at all



TRUE if e is present!

Temporal logic connector

Sequences can be composed by concatenation. The concatenation specifies a delay, using ##. It is used as follows:

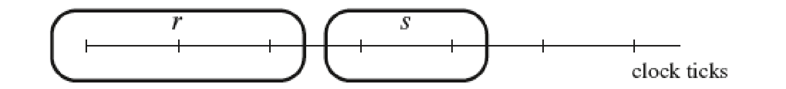
> ## integral_number or identifier
>
> ## (constant_expression)
>
> ## [cycle_delay_const_range_expression]

cycle_delay_const_range_expression := **const:const** or **const:$**

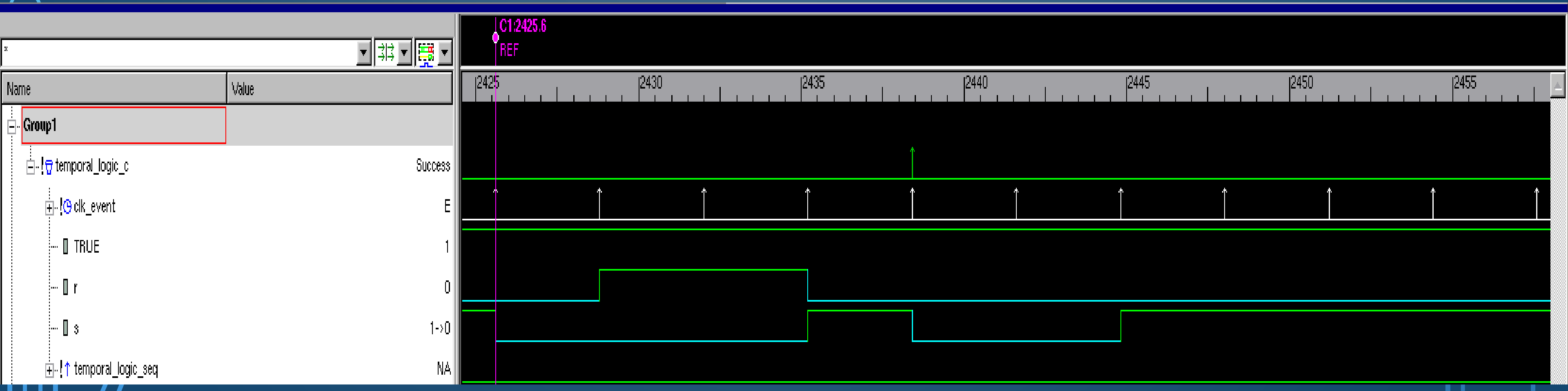
$ represents a non-zero and finite number

Example:

> r ##1 s



There is a match of sequence "r ##1 s" if there is a match of sequence **r** and there is a match of sequence **s** starting from the clock tick immediately following the match of **r**

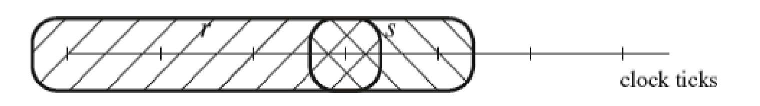


Name	Value
Group1	
temporal_logic_c	Success
clk_event	E
TRUE	1
r	0
s	1->0
temporal_logic_seq	NA

Sequence fusion:

r ##0 s

The fusion of sequences **r** and **s**, is matched if for some match of sequence **r** there is a match of sequence **s** starting from the clock tick where the match of **r** happened



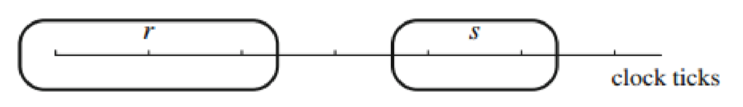
Multiple Delays:

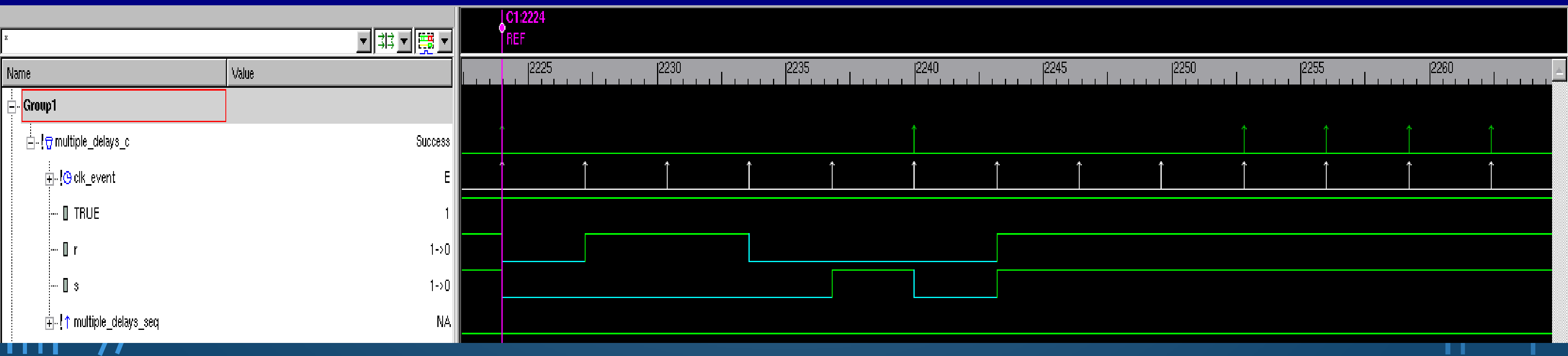
r ##n s

r is true on current tick, **s** will be true on the **n**th tick after **r**

Example:

r ##2 s

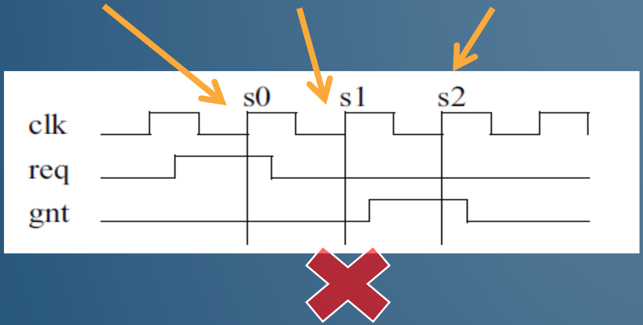




Name	Value
Group1	
multiple_delays_c	Success
clk_event	E
TRUE	1
r	1->0
s	1->0
multiple_delays_seq	NA

Example:

req ##1 gnt ##1 !req

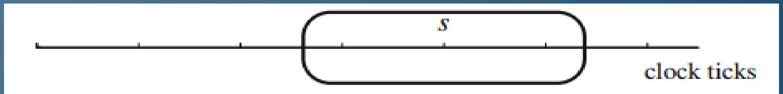


Initial Delay:

##n s

Specify the number of clock ticks to be skipped before beginning a sequence match.

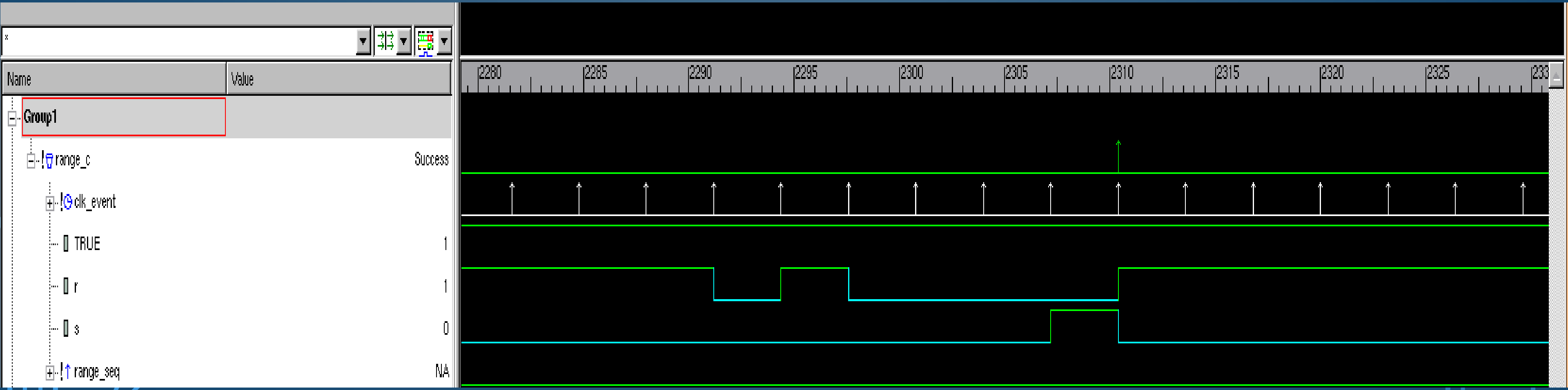
Example: ##3 s



Range:

r ##[M : N] s

means that if **r** is true on current tick, **s** will be true M to N ticks from current tick



Name	Value
Group1	
range_c	Success
clk_event	
TRUE	1
r	1
s	0
range_seq	NA

Example:

a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b

by simplification the previous sequence results in:

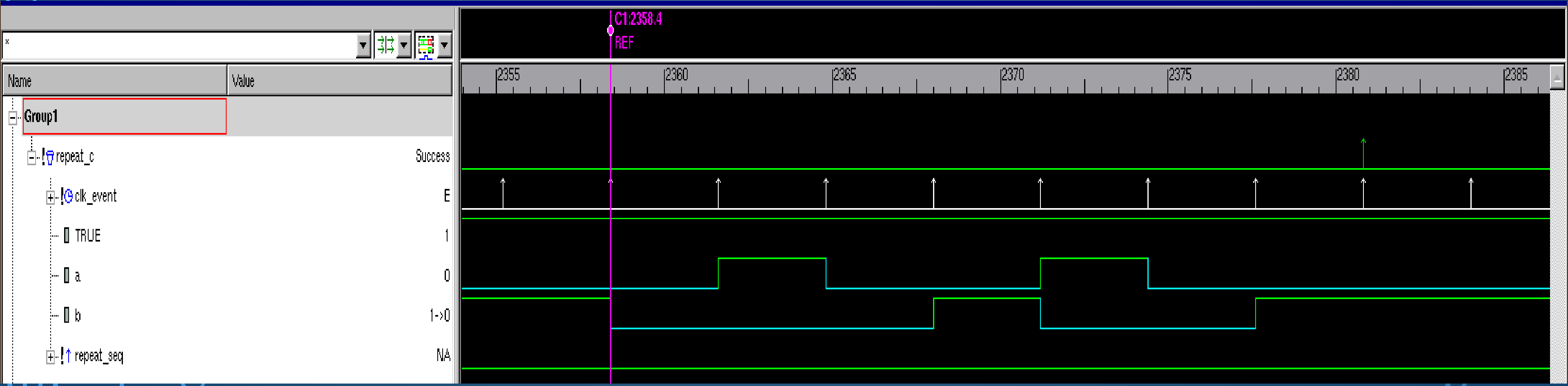
(a ##2 b) [*5]

r ##[*M : N] s

Repeat **r** at least M times and as many as N times consecutively

r ##[*M : $]

Repeat **r** an unknown number of times but at least M times



Name	Value
Group1	
repeat_c	Success
clk_event	E
TRUE	1
a	0
b	1->0
repeat_seq	NA

Go to Repetition:

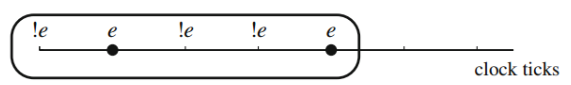
r ## 1 s [->N] ##1 t

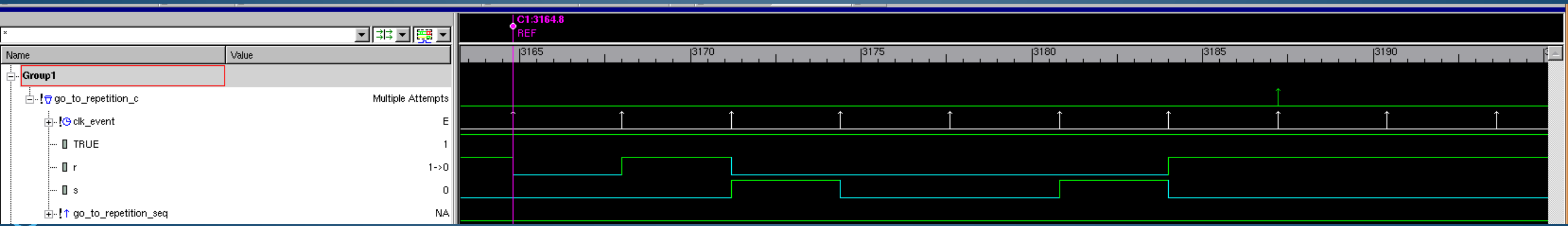
Means **r** followed by exactly N not necessarily consecutive **s**'s with last **s** followed the next tick by t

r ##1 s [->M : N] ##1 t

Means **r** followed by at least **M**, at most **N s**'s followed next tick by **t**

Example: e [->2]





Name	Value
Group1	
go_to_repetition_c	Multiple Attempts
clk_event	E
TRUE	1
r	1->0
s	0
go_to_repetition_seq	NA

Non-Consecutive Repetition

r ## 1 s [= N] ##1 t

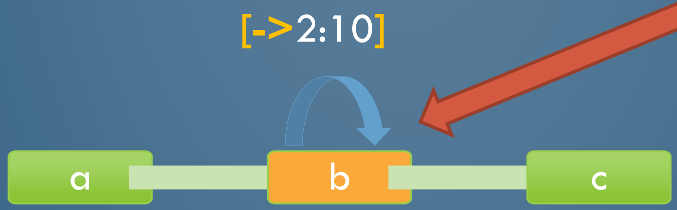
Means **r** followed by exactly N not necessarily consecutive **s**'s with last **s** followed sometime by **t**

r ##1 s [= M : N] ##1 t

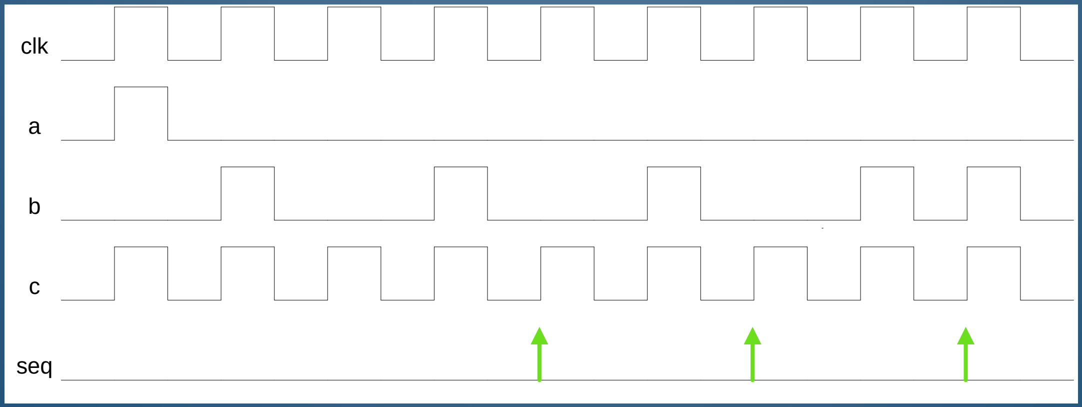
Means **r** followed by at least **M**, at most **N s**'s followed some time by **t** "**t** does not have to follow immediately after the last **s**"

What does the following sequence mean?

a ##1 b [->2:10] ##1 c



Watch out for the number of threads!



How can we interpret the following sequence?

a ##1 b [=2:10] ##1 c

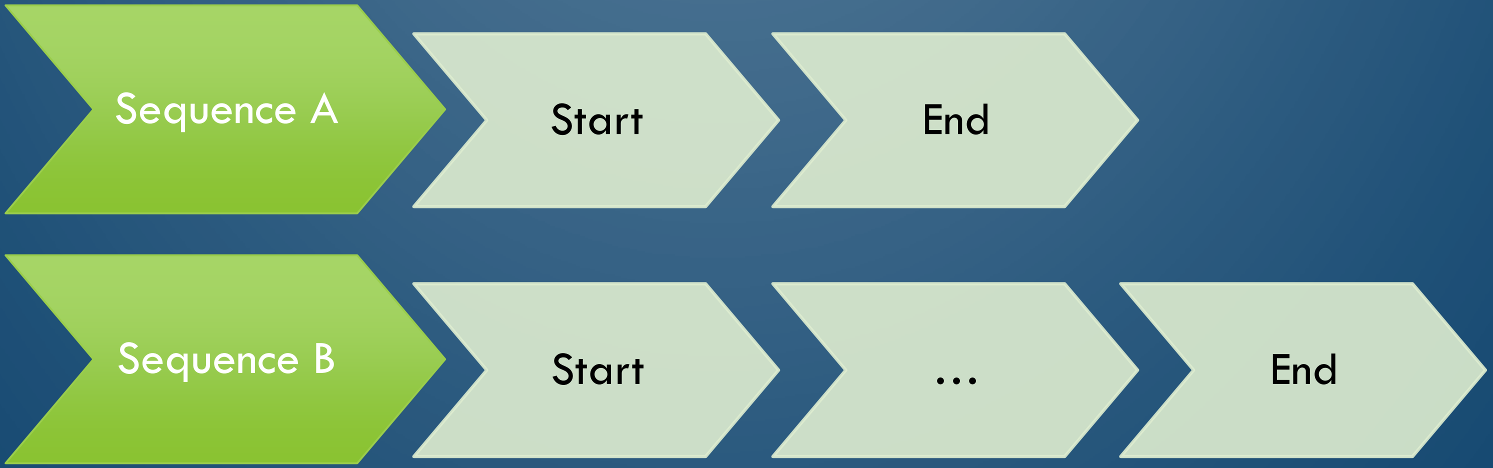
And

The binary operator **and** is used when both operands are expected to match, but the end times of the operand sequences can be different.

It is used as follows:

Sequence A **and** Sequence B

where both operands must be sequences.



Sequence A

Start

End

Sequence B

Start

…

End

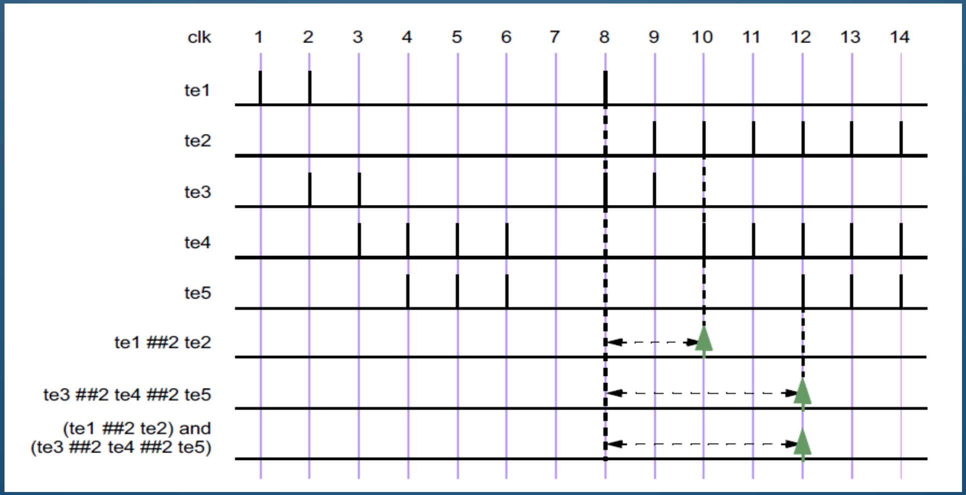
One can say:

a) The operand sequences start at the same time.
b) When one of the operand sequences matches, it waits for the other to match.
c) The end time of the composite sequence is the end time of the operand sequence that completes last.

Example:

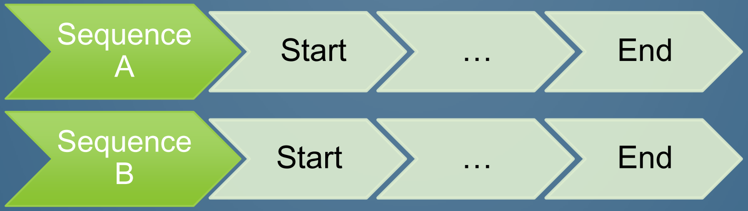
(te1 ##2 te2) **and** (te3 ##2 te4 ##2 te5)

What if the two operands are Boolean expresions? How does the and operation behave?



Intersect

The binary operator **intersect** is used when both operand sequences are expected to match, and the end times of the operand sequences must be the same. It is used in the same way as the and operation.

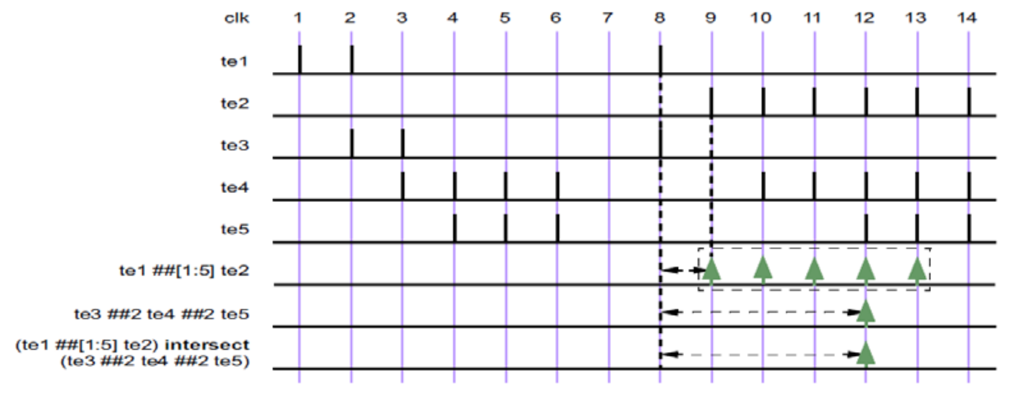

Sequence A | Start | … | End

Sequence B | Start | … | End

One can conclude that the additional requirement on the length of the sequences is the basic difference between **and** operation and **intersect** operation.

Example:

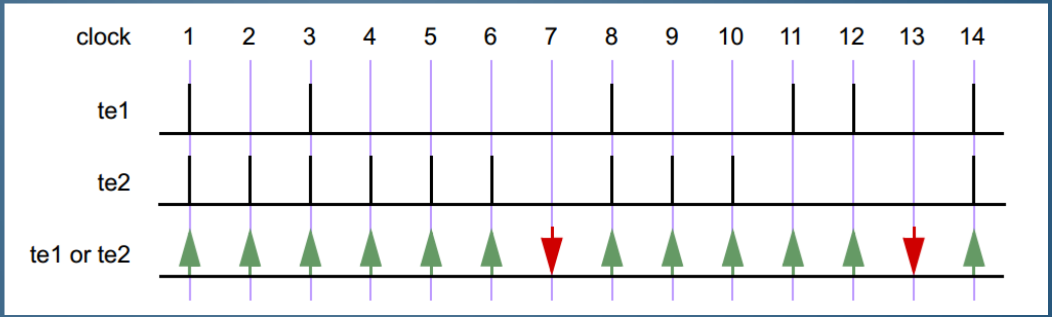
(te1 ##[1:5] te2) **intersect** (te3 ##2 te4 ##2 te5)



Or

The operator **or** is used when at least one of the two operand sequences is expected to match. It is used in the same way as the and operation [1].

te1 or te2



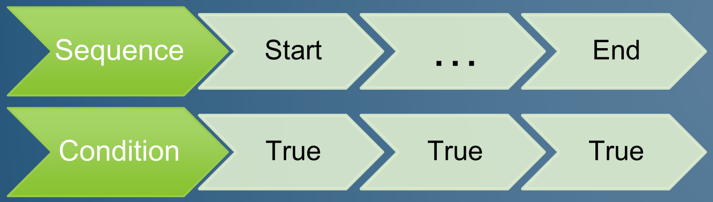
Throughout

Sequences often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction.

It is used as follows:

expression_or_dist **throughout** sequence_expr

where an expression or distribution must hold true during the whole sequence.

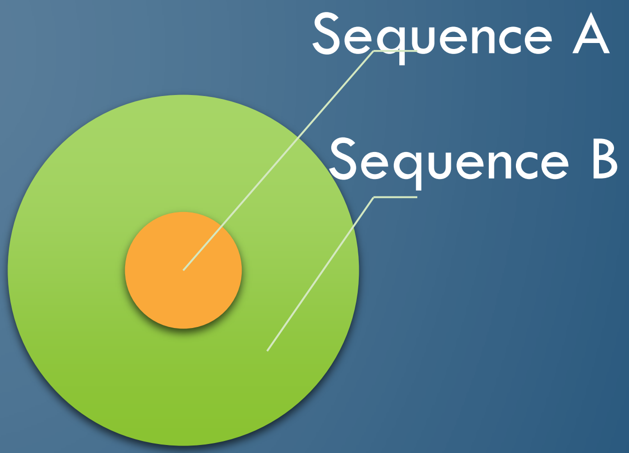


One can understand the throughout condition as two processes that run in parallel.

Within

The containment of a sequence within another sequence is expressed with the **within** operation. This condition is used as follows:

(sequence_expr) **within** (sequence_expr)



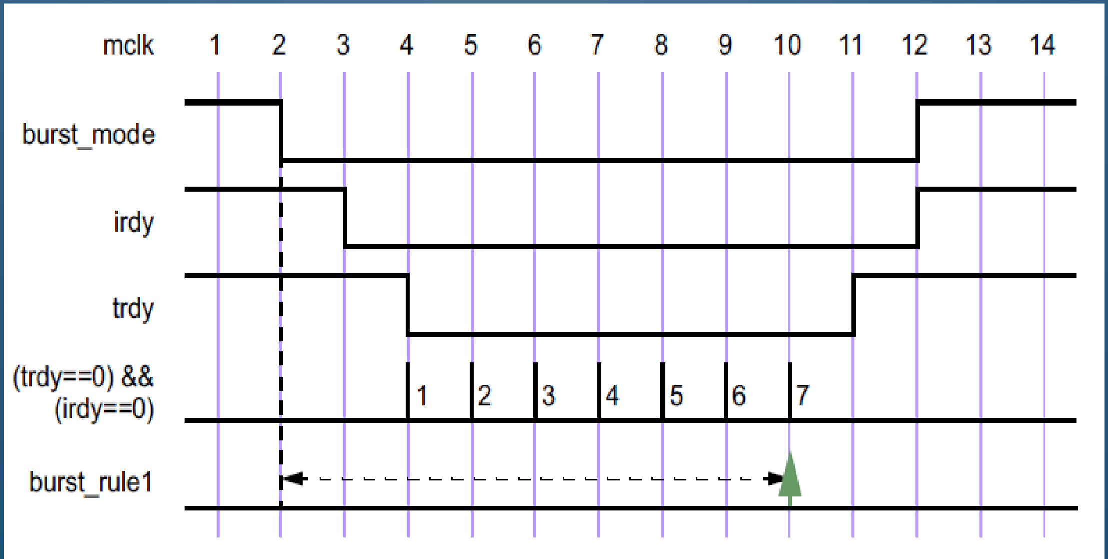
One can conclude that:

a) The start point of the match of seq1 must be no earlier than the start point of the match of seq2.
b) The end point of the match of seq1 must be no later than the end point of the match of seq2.

ADDITIONAL SEQUENCE FEATURES

How can we describe the following condition?

```
!trdy [*7]     within ( $fell(irdy)   ##1 !irdy[*8] )
```



I. Detecting and using end point of a sequence could ease to describe a complex sequence that uses the first as a start point.

Example:

```
sequence s;
     a ##1 b ##1 c;
endsequence

sequence rule;
   @(posedge sysclk)  trans ##1 start_trans ##1 s.ended ##1  end_trans;
endsequence
```

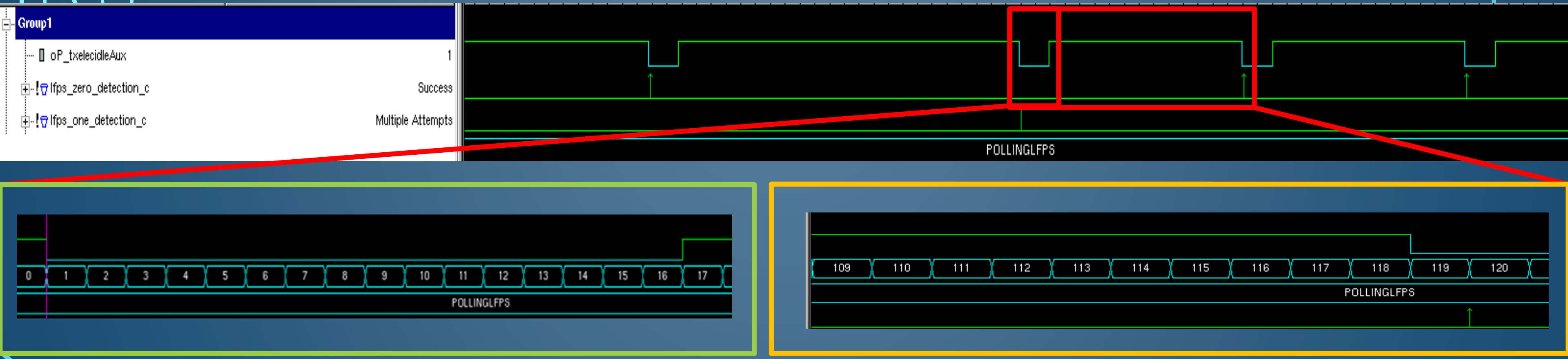
II. Manipulating data in a sequence.

Example:

```
   sequence add_3;
     a ##1 ( b[->1], x = pipe_in) ##1 c[*2] ##0 ( pipe_out == x + 3);
   endsequence
```

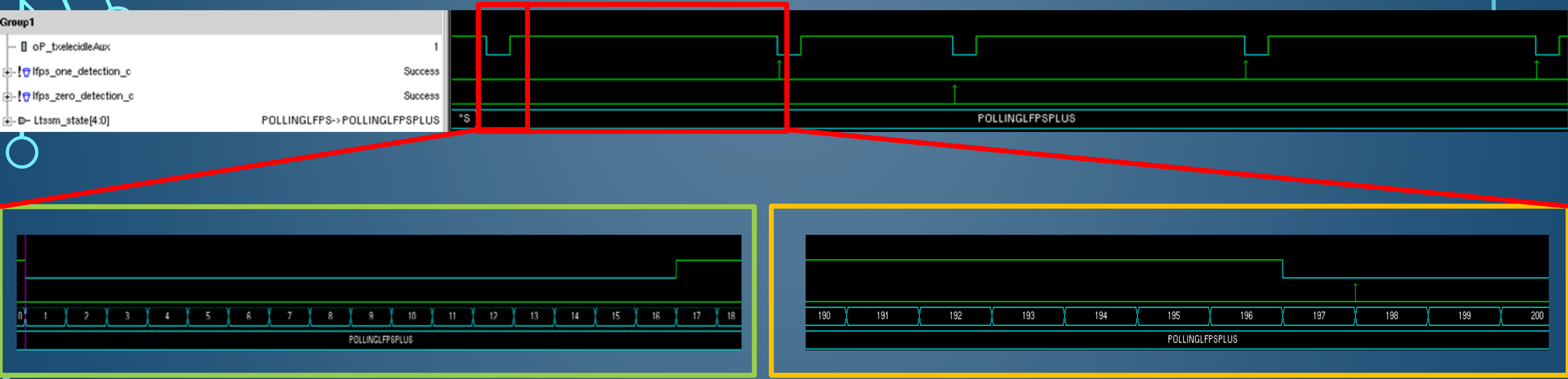
USB Examples

USB3.1 LFPS Zero Detection



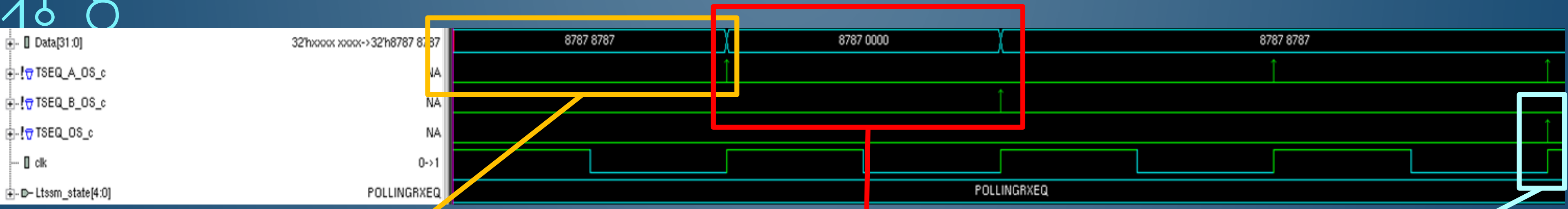
```
// LOW_DURATION_1:16        LOW_DURATION_2:18
// HIGH_DURATION_1:102      HIGH_DURATION_2:104

sequence lfps_zero_detection_c;
  @(posedge clk)(
   (!oP_txelecidleAux) [*LOW_DURATION_1:LOW_DURATION_2]   ##1
   (oP_txelecidleAux)  [*HIGH_DURATION_1:HIGH_DURATION_2] ##1
   (!oP_txelecidleAux)
  );
Endsequence : lfps_zero_detection_c
```

USB3.1 LFPS ONE DETECTION



```
// LOW_DURATION_1:16        LOW_DURATION_2:18
// HIGH_DURATION_1:180      HIGH_DURATION_2:184

sequence lfps_one_detection_c;
  @(posedge clk)(
   (!oP_txelecidleAux) [*LOW_DURATION_1:LOW_DURATION_2]   ##1
   (oP_txelecidleAux)  [*HIGH_DURATION_1:HIGH_DURATION_2] ##1
   (!oP_txelecidleAux)
  );
Endsequence : lfps_one_detection_c
```

USB3.1 TSEQ DETECTION



```
// TSEQ_A_SEQUENCE: 87878787

sequence tseqA_detection_seq;
  @(posedge clk)(
    Data == TSEQ_A_SEQUENCE
  );
endsequence
```

```
// TSEQ_B_SEQUENCE: 87870000

sequence tseqB_detection_seq;
  @(posedge clk)(
    Data == TSEQ_B_SEQUENCE
  );
endsequence
```

```
sequence tseq_detection_seq;
  @(posedge clk)(
    tseqA_detection_seq ##1
    tseqB_detection_seq ##1
    tseqA_detection_seq ##1
    tseqA_detection_seq
  );
endsequence
```

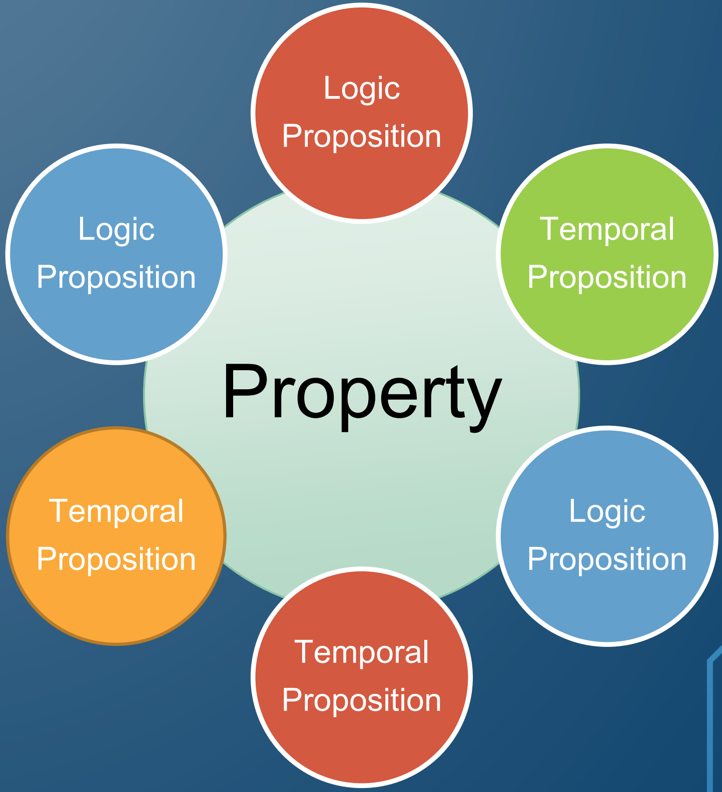
IV - PROPERTY

Basics

Definition:
A property defines an abstract behavior of the design. The result of property evaluation is either true or false.

❑ The property definition is based on propositional and temporal logic which deal with simple declarative propositions or simple declarative propositions through time respectively.

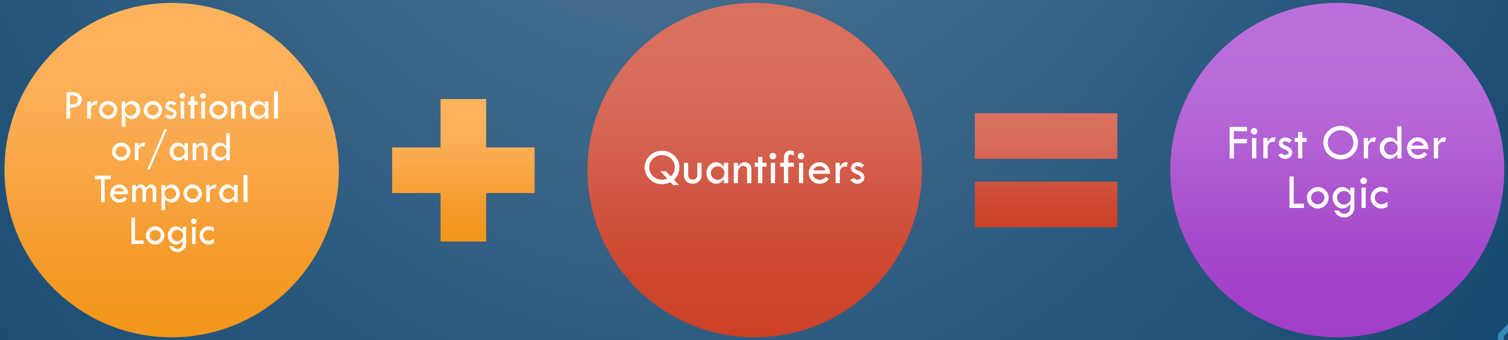
❑ Note: The combination of some propositional/temporal logic elements with **generate for** can leads to first-order logic which covers predicates and quantification.



- A predicate resembles a function that returns either True or False.

- First-order logic allows reasoning about properties that are shared by many objects, through the use of variables.

- First-order logic is distinguished from propositional logic by its use of quantifiers; each interpretation of first-order logic includes a domain of discourse over which the quantifiers range.



❑ In SV a **property** can be declared in:

- a module,
- an interface,
- a program,
- a clocking block,
- a package and a compilation-unit scope [1].

A **property** declaration by itself does not produce any result.

❑ There are seven kinds of properties: sequence, negation, disjunction, conjunction, if...else, implication, and instantiation (reusable properties).

A **property** declaration is as follows:

```
property rule6_with_no_type(x, y);
        ##1 x |-> ##[2:10] y;
endproperty : rule6_with_no_type
```

Property Construction

Property Type: Sequence

A property that is a sequence evaluates to true if, and only if, there is a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property.

Example:

```
property prop_seq;
  @(posedge clk)  $rose(rqst) ##1 $rose(gnt);
Endproperty : prop_seq
```

Property Type: Negation

For each evaluation attempt of the property, there is an evaluation attempt of *property_expr*. The keyword **not** states that the evaluation of the property returns the opposite of the evaluation of the underlying *property_expr*.

Example:

```
property prop_not;
   @(posedge clk) not property_expr;
endproperty : prop_not
```

Property Type: Disjunction

A property is a disjunction if it has the form:

property_expr1 **or** *property_expr2*

The property evaluates to true if, and only if, at least one of *property_expr1* and *property_expr2* evaluates to true.

Property Type: Conjunction

A property is a conjunction if it has the form:

property_expr1 **and** *property_expr2*

The property evaluates to true if, and only if, both *property_expr1* and *property_expr2* evaluate to true.

Property Type: If ... Else

A property is an if...else if it has either the form:

if (expression_or_dist) *property_expr1*

or the form

if (expression_or_dist) *property_expr1* **else** *property_expr2*

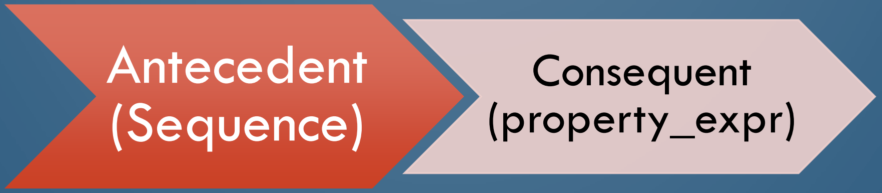
A property of the first form evaluates to true if, and only if, either *expression_or_dist* evaluates to false or property_expr1 evaluates to true.

A property of the second form evaluates to true if, and only if, either *expression_or_dist* evaluates to true and *property_expr1* evaluates to true or *expression_or_dist* evaluates to false and *property_expr2* evaluates to true.

Property Type: Implication

The implication construct specifies that the checking of a property is performed conditionally on the match of a sequential antecedent.

This clause is used to precondition monitoring of a property expression and is allowed at the property level. The result of the implication is either true or false.



Antecedent (Sequence)

Consequent (property_expr)

Two forms of implication are provided: overlapped using operator **|->** and non-overlapped using operator **|=>**. Therefore, a property is an implication if it has either the form (non-temporal)

sequence_expr **|->** property_expr

or the form (temporal)

sequence_expr **|=>** property_expr

Property Type: Instantiation

An instance of a named property can be used as a *property_expr or property_spec*.

In general, the instance is legal provided the body *property_spec* of the named property can be substituted in place of the instance, with actual arguments substituted for formal arguments, and result in a legal *property_expr* or *property_spec,* ignoring local variable declarations.

Example

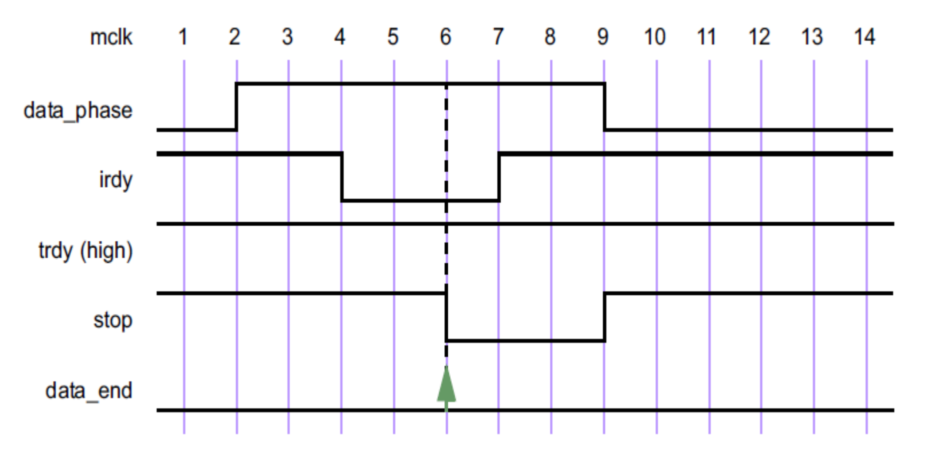
I. Objective: Data Transfer Master → Target Bus Operation

II. Functional Details:

- Data Transfer includes multiple data phases
- Data phase completes on rising edge of clk when irdy && (trdy || stop)
- All signals are active low

The end of a data phase can be expressed as follows:

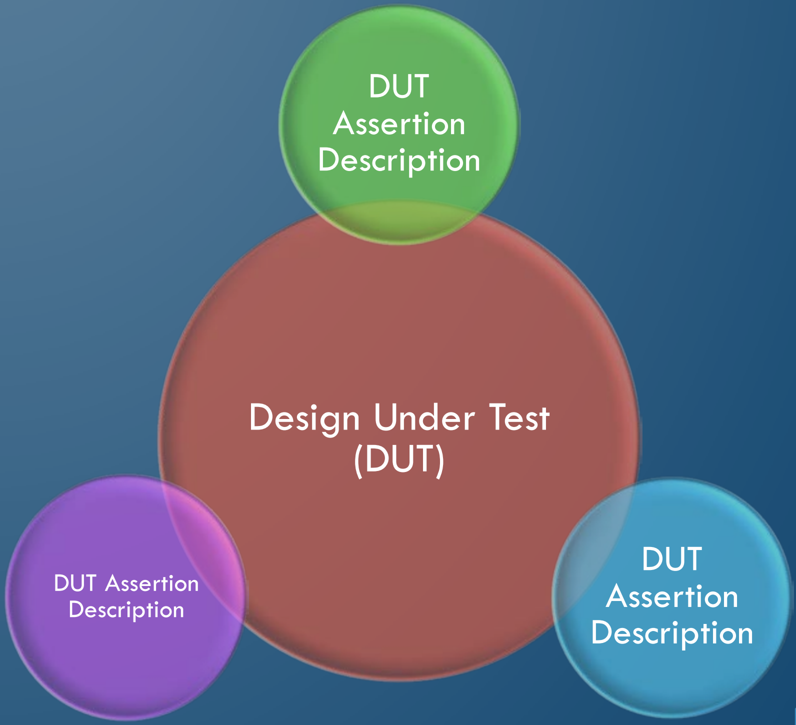
```
property data_end;
 @(posedge mclk) data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop)));
endproperty
```



V. ASSERTION LANGUAGE

INTRODUCTION

Definition : The assertion language is used to specify the correct behavior of the system under consideration.


DUT Assertion Description
Design Under Test (DUT)
DUT Assertion Description
DUT Assertion Description

Assertions are used to express the design intent. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation. [1]

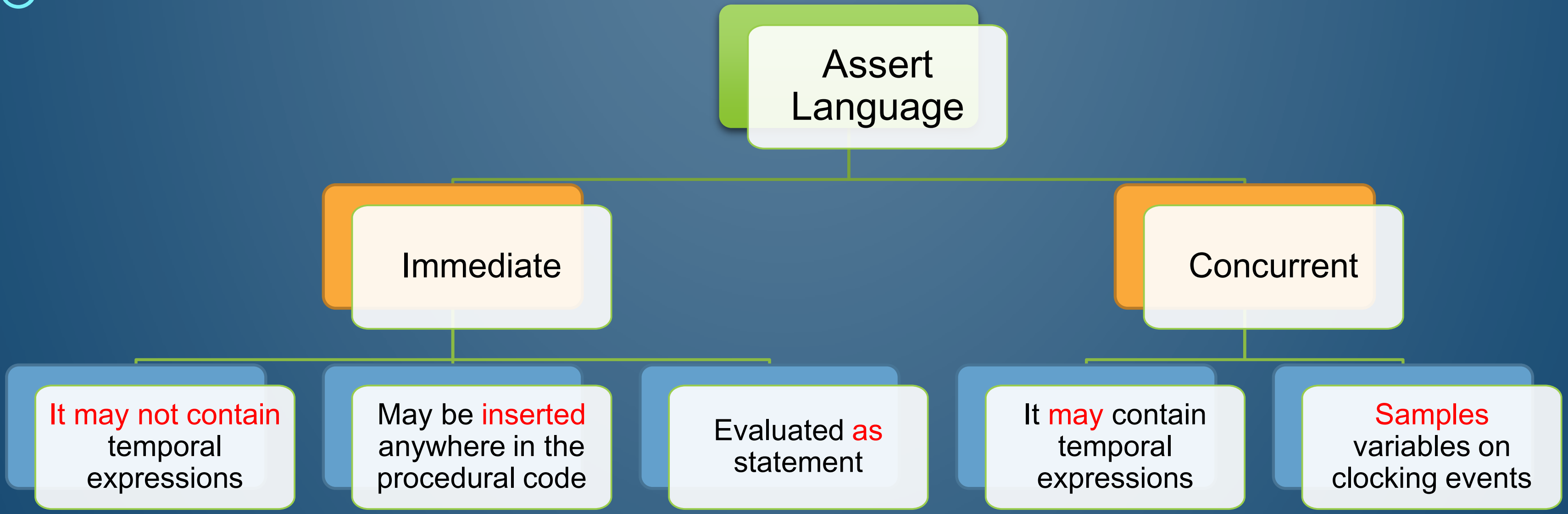
- ❑ By covering properties one can check if a certain design specification was stimulated (functional coverage).
- ❑ When the model is restricted to certain assumptions the input stimulus are restricted (generated) as well, i.e. using properties inside constraint blocks to restrict random stimulus generation [1].

With SVA a timing accurate input/output model description for a design (**what, when**) can be done, without providing any details about **how** the job is done.

ASSERT LANGUAGE

- **Immediate assertions:** Follow simulation event semantics for their execution and are executed like a statement in a procedural block [1].

- **Concurrent assertions:** This assertions are based on clock semantics and use sampled values of variables. This simplify the evaluation of a circuit description [1].



Assert Language

Immediate

Concurrent

It may not contain temporal expressions

May be inserted anywhere in the procedural code

Evaluated as statement

It may contain temporal expressions

Samples variables on clocking events

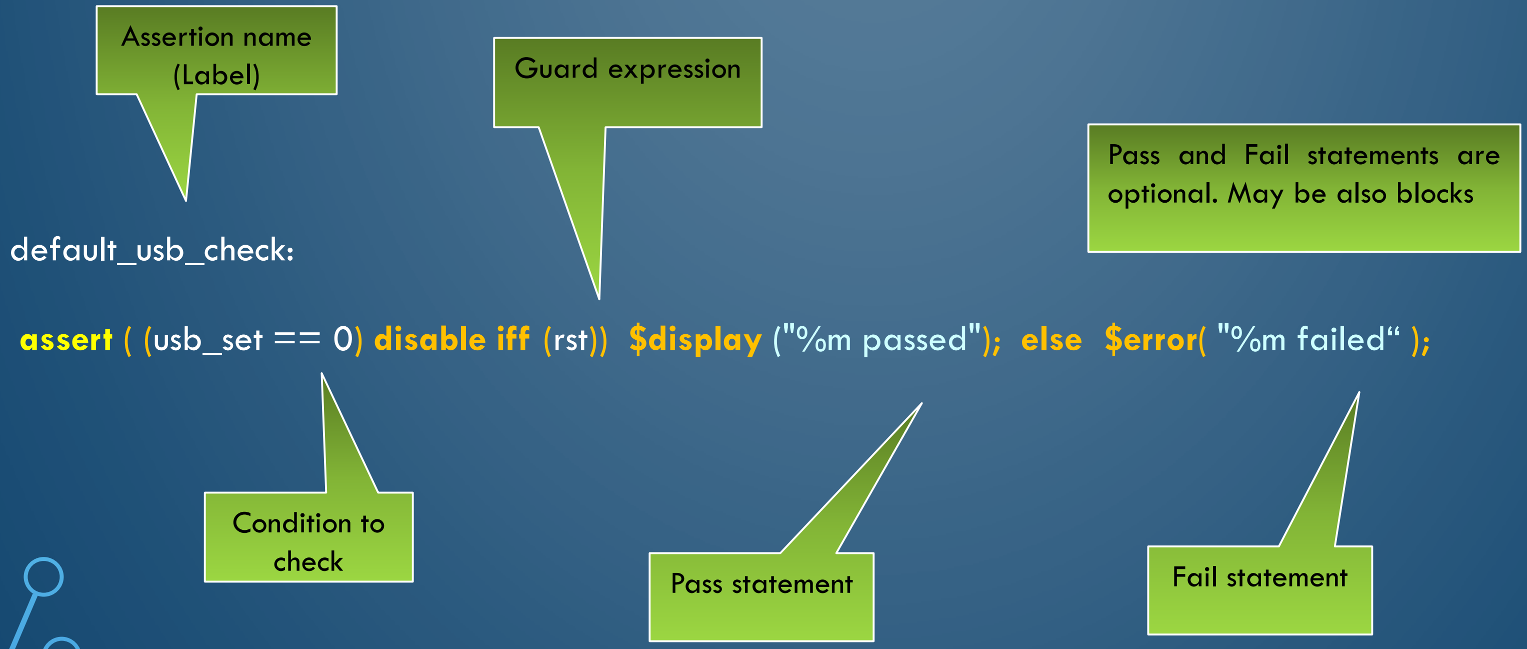
IMMEDIATE ASSERTIONS

If the non-temporal expression evaluates to X, Z, or 0, then it is interpreted as being false, and the assertion is said to fail. Otherwise, the expression is interpreted as being true, and the assertion is said to pass.

SystemVerilog syntax:

[*label*:] **assert (** <*immediate_property*> [**disable iff** <*disable_condition*>] **)** <*action_block*>
disable_condition := expression
immediate_property := non_temporal_logic_expression **|** non_temporal_property_name
action_block := statement_or_null [**else** statement]

Example:



Assertion name (Label)

Guard expression

Pass and Fail statements are optional. May be also blocks

```
default_usb_check:

 assert ( (usb_set == 0) disable iff (rst))  $display ("%m passed");  else  $error( "%m failed“ );
```

Condition to check

Pass statement

Fail statement

Severity System Tasks

Because the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is *error.*

Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

- ❑ **$fatal** is a run-time fatal.
- ❑ **$error** is a run-time error.
- ❑ **$warning** is a run-time warning, which can be suppressed in a tool-specific manner.
- ❑ **$info** indicates that the assertion failure carries no specific severity.

The severity system tasks use the same syntax as $display and they can be used with both immediate and concurrent assertions

ii. CONCURRENT ASSERTIONS

Temporal

- Describe behavior that spans over time.
- The evaluation model is based on a clock.

The values of variables used in the evaluation are the sampled values.

- A predictable result can be obtained from the evaluation.

SystemVerilog syntax:

[*label*:] **assert property (** *property_spec* **)** action_block

See full description [1, A.2.10]

property_spec ::= [clocking_event] [**disable iff (** expression_or_dist **)**] property_expr

Example:

Assertion name (Label)

Sampling event

Asynchronous reset

Assertion body

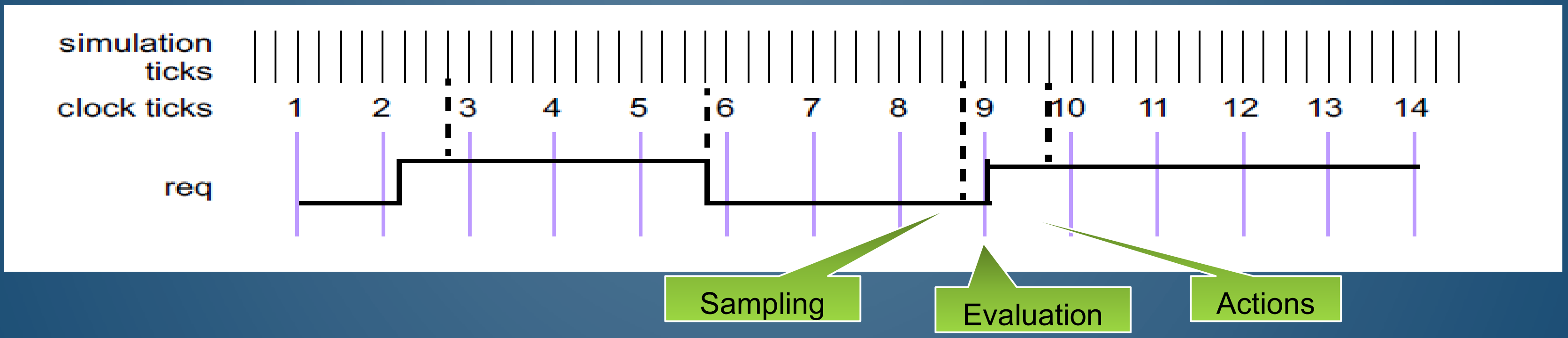
```
my_concurrent_check:

assert property ( @ ( posedge clk ) disable iff ( rst ) not ( a ##1 b ))
$info ( “Property p passed” );  else  $error ( “Property p failed”);
```

Pass statement

Fail statement

SAMPLING

The values of variables used in assertions are sampled in the Preponed* region of a time slot, and the assertions are evaluated during the Observe* region. Action blocks are scheduled in Reactive region.



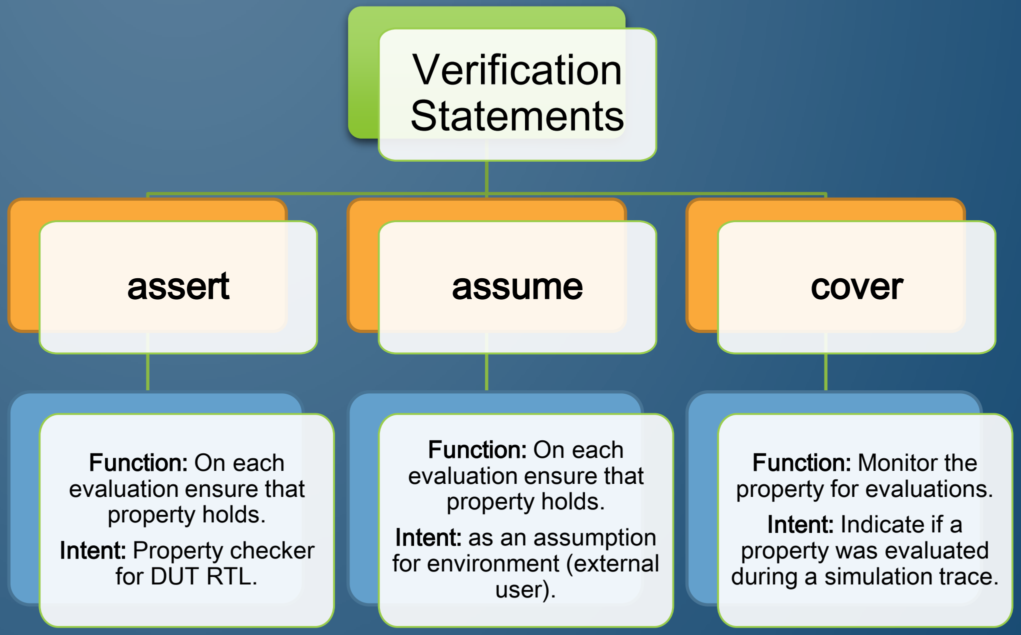
For concurrent assertions, the following statements apply:

- ❑ It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.
- ❑ If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

* See [1, chap 9]

VERIFICATION STATEMENTS

A property on its own is never evaluated, it must be used within a verification statement for this to occur. A verification statement states the verification function (intent) to be performed on the property.



Verification Statements

assert

Function: On each evaluation ensure that property holds.

Intent: Property checker for DUT RTL.

assume

Function: On each evaluation ensure that property holds.

Intent: as an assumption for environment (external user).

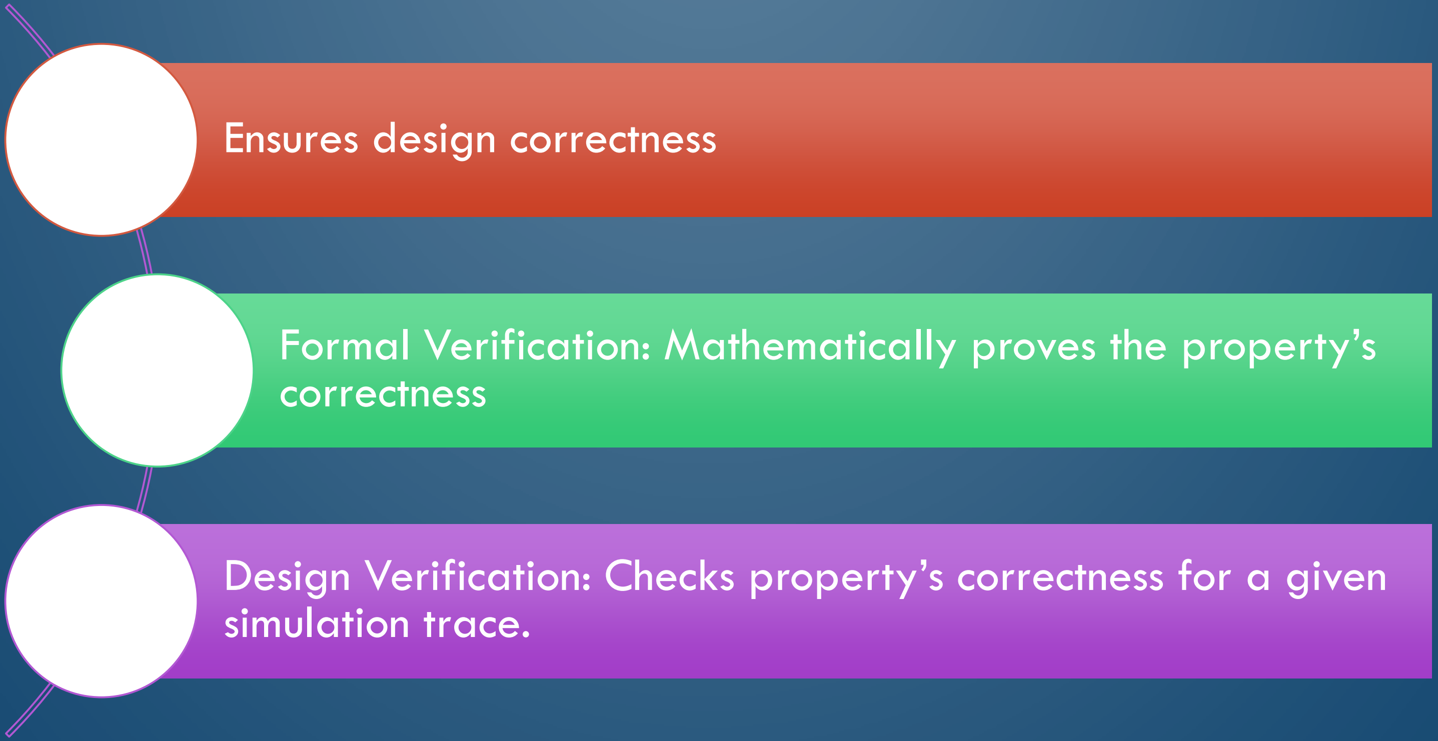
cover

Function: Monitor the property for evaluations.

Intent: Indicate if a property was evaluated during a simulation trace.

Assert

The purpose of the **assert** statement is to check the equivalence between the abstract description (property) and the functional description (RTL) during formal analysis and dynamic simulations.



Ensures design correctness

Formal Verification: Mathematically proves the property's correctness

Design Verification: Checks property's correctness for a given simulation trace.

The assert statement follows this syntax:

```
assert property ( property_spec ) action_block
```

See full description [1, A.2.10]

Example:

```
property abc(a,b,c);
   disable iff (a == 2) @(posedge clk) not (b ##1 c);
endproperty

env_prop: assert property ( abc ( rst,in1,in2 ) )
$display( “env_prop passed.“ ); else $display( “ env_prop failed.“ );
```

Assume

The purpose of the **assume** statement is to allow properties to be considered as assumptions (oriented to external drivers/responders) for formal analysis as well as for dynamic simulation tools.

Specifies requirements for the environment.

Design Verification

- It is treated the same as assertions.
- There is no requirement on the tools to report successes of the assumed properties.

Formal Verification

- Restricts the model.
- The property is considered as a hypothesis to prove the asserted properties

The assume statement follows this syntax:

```
assume property ( property_spec ) ;
```

See full description [1, A.2.10]

No action blocks!!!
but messaging is allowed

Example:

A simple synchronous request and acknowledge protocol, where variable *req* can be raised at any time and must stay asserted until *ack* is asserted. In the next clock cycle, both *req* and *ack* must be deasserted.

Properties governing *req* are as follows:

```
property pr1;
   @( posedge clk )
   !reset_n |-> !req;     // when reset_n is asserted (0),keep req 0
endproperty
```

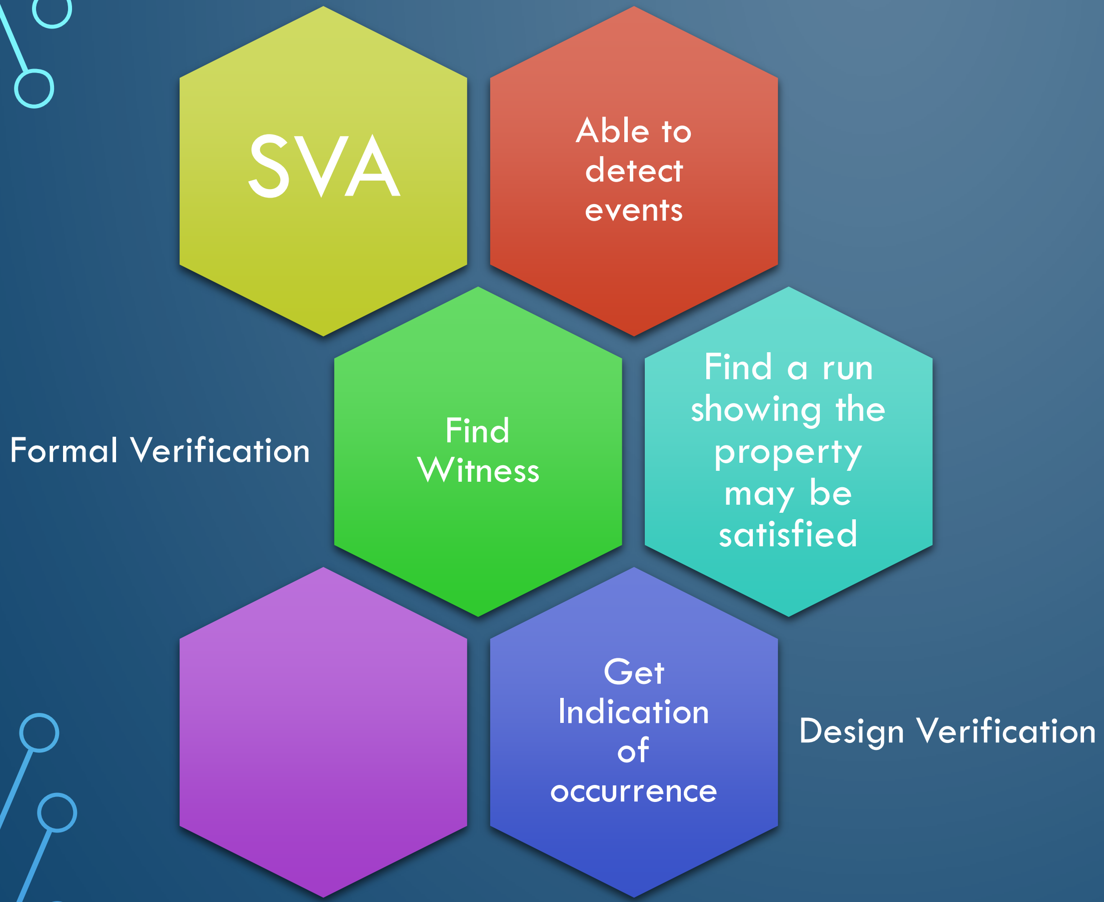
```
property pr2;
    // one cycle after ack, req must be deasserted
    @(posedge clk) ack |=> !req;
endproperty
```

```
property pr3;
     // hold req asserted until and including ack asserted
     @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

The following properties are assumed:

```
assume_req1: assume property (pr1);
assume_req2: assume property (pr2);
assume_req3: assume property (pr3);
```

Cover


SVA
Able to detect events
Formal Verification
Find Witness
Find a run showing the property may be satisfied
Get Indication of occurrence
Design Verification

- ❑ The purpose of the **cover** is to monitor properties of the design for coverage, i.e. to count the number of times a property was evaluated (disregarding the result of the evaluation).

- ❑ The tools can gather information about the evaluation and report the results at the end of simulation.

The cover statement follows this syntax:

```
cover property ( property_spec ) statement_or_null
```

See full description [1, A.2.10]

Example:

```
cover property ( @ ( posedge clk ) !rst  throughout req ##1  ack  );
```

VI. COMPARATIVE RESULTS

Formal Temporal Logic vs Scripting

- Just comparing code to represent the sequence model:
  - ~3.5x gain
- Considering the built of coverage, assertion and the sequence encapsulation:
  - ~1.7x gain

Formal Temporal Logic vs Structural Modeling (Scoreboard)

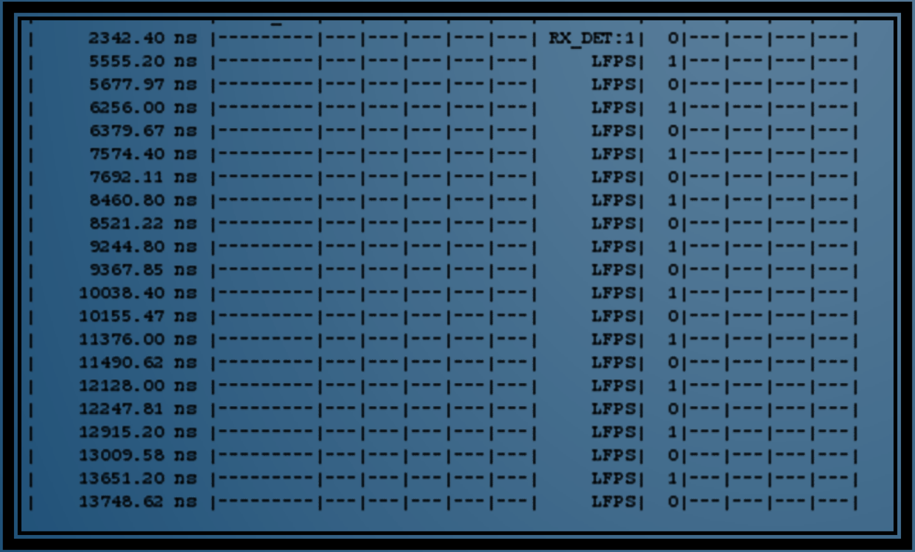
- Just comparing code to represent the sequence model:
  - ~6x gain
- Considering the built of coverage, assertion and the sequence encapsulation:
  - ~3x gain

+

Additional Multiplication Gain Factor:
Each time a library sequence is used!!!

Figure of Merit: Number of Code Lines

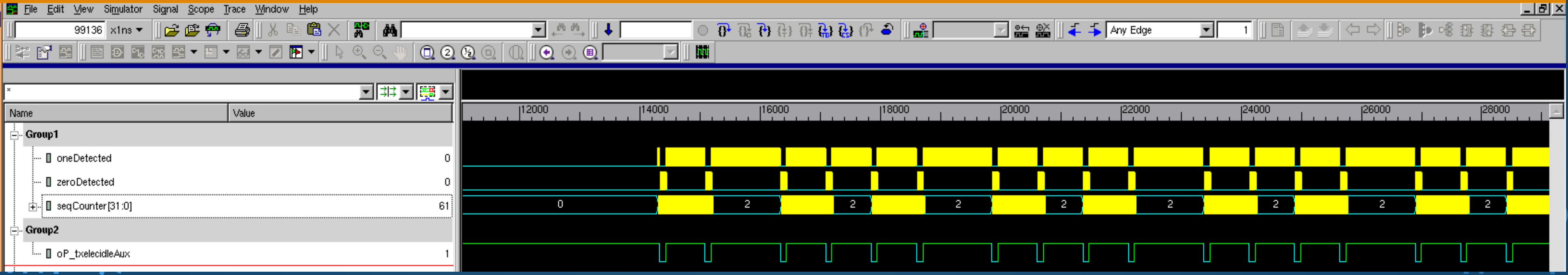
CODING EFFORT RESULTS



```
|     2342.40 ns |---------|---|---|---|---|---| RX_DET:1|  0|---|---|---|---|
|     5555.20 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|     5677.97 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|     6256.00 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|     6379.67 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|     7574.40 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|     7692.11 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|     8460.80 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|     8521.22 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|     9244.80 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|     9367.85 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|    10038.40 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|    10155.47 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|    11376.00 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|    11490.62 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|    12128.00 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|    12247.81 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|    12915.20 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|    13009.58 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
|    13651.20 ns |---------|---|---|---|---|---|     LFPS|  1|---|---|---|---|
|    13748.62 ns |---------|---|---|---|---|---|     LFPS|  0|---|---|---|---|
```

Perl required lines to Process a simple sequence of 1's and 0's:
41 Code Lines

SV required lines to process a simple sequence of 1's and 0's without using "temporal" logic:
74 Code Lines

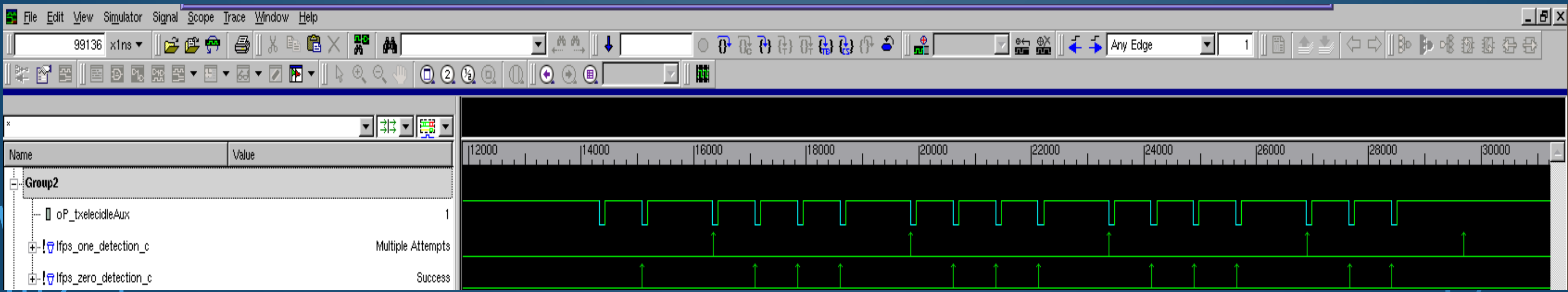


Name	Value
Group1	
oneDetected	0
zeroDetected	0
seqCounter[31:0]	61
Group2	
oP_txelecidleAux	1

Perl VS. Formal temporal logic (using SystemVerilog) : 2.5 x less coding effort

<u>Using formal temporal logic </u>to process a simple sequence of 1's and 0's: 16 Code Lines!

Verilog VS. Formal temporal logic (using SystemVerilog): 4.6 x less coding effort



Name	Value
Group2	
oP_txelecidleAux	1
lfps_one_detection_c	Multiple Attempts
lfps_zero_detection_c	Success

THANK YOU!