SVA Advanced Topics: SVAUnit and Assertions for Formal
SystemVerilog Assertions
Verification with SVAUnit

Andra Radu
Ionuț Ciocîrlan

AMIQ Consulting
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?

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

• What is a property?

```plaintext
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

\[
\text{assert (expression) pass}\_\text{statement} \\
\quad \text{[else fail}\_\text{statement]}
\]

- Concurrent

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

\[
\begin{align*}
\text{req} & \\
\text{ack} & \\
\text{SVA state} & \\
0 & 1 & 2 & 3 & 4 & 5
\end{align*}
\]

- SVA succeeded
- SVA failed
 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 \( \rightarrow \) 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)
  );
  ```
• Avoid overlapping assertions that contradict each other

✓ CPU_0: \texttt{assert property (WRITE } \implies \text{ERROR);}

✓ CPU_1: \texttt{assert property (WRITE } \implies \neg \text{ERROR);}

\texttt{assert property (WRITE and CPU==0 } \implies \text{ERROR);}

\texttt{assert property (WRITE and CPU==1 } \implies \neg \text{ERROR);}
Best Practices (contd.)

• Use the $sampled() function in action blocks

```verilog
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)

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

• Consequent (the ACCESS phase)

```verilog
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:

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

• The assertion will look like:

```verilog
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

Adaptable | Self-contained | Predictable
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

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

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:

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

• The assertion will look like:

```vhdl
assert property (access_to_setup_p)
else `uvm_error("ERR", "Assertion failed")
```
Example of SVAUnit Testbench

``` verilog
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 utl 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!"")
   endfunction

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

   task test();
      // Initialize signals
      // Create scenarios for SVA verification
   endtask
endclass
APB – SVAUnit Test Steps

1. Enable the APB SVA
2. Initialize the interface signals
3. Generate setup phase stimuli
4. Generate access phase stimuli
5. SVA checks based on generated stimuli
Enable SVA and Initialize Signals

... 

// Enable the APB SVA 
scanf.disable_all_assertions();
scanf.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

...
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

...
... 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_allAssertions();
- 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

1. Create SVAUnit Testbench
2. Create an SVAUnit Test
3. Implement test() task
4. Create an SVAUnit Test Suite
5. Instantiate test in Test Suite
6. Register tests in test suite
7. Simulate
8. Scan report
Error Reporting

- **Name of SVAUnit check**
- **SVAUnit test path**

```
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 SVA under test**
- **Custom error message**
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

- \( a \text{ and } b \implies c \)

```
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

- $\$rose(a) \#\#[1:4] b \rightarrow \#\#[1:3] c$

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

```verilog
class 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

- $a \implies b[*1:2] \#1 c$

```vhdl
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.)

• \( a \Rightarrow b^{*1:2} \) \#\#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

• $a \implies b[*0:2] \#1 \ c$

```plaintext
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.)

- $a \rightarrow b[*0:2] \#1 c$

```plaintext
...  // (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

\[ a \implies (b \ #\#\# c) \text{ or } (d \ #\#\# e) \]

```
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 \ #\#\# c)
            SVA state check based on branch stimuli
        end
        begin
            Stimuli for branch: (d \ #\#\# e)
            SVA state check based on branch stimuli
        end
    join
end
```
Sequence Disjunction (contd.)

- \( a \implies (b \text{ or } c) \text{ or } (d \text{ or } e) \)

```verilog

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
```

...
• a \implies (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.)

• \( a \implies (b \equiv_1 c) \text{ or } (d \equiv_1 e) \)

```vhdl
// 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

• 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!
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.
     - Temporal logic connectors
     - 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 Assertions (SVA) is unambiguous:

```systemverilog
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.
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

Boolean Expressions

Sequences

Properties

Implication of Sequences

Assertion Statements

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

A single tree branch can be understood as a realization of one possible path through time.
II - LANGUAGE CONSTRUCTS
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 \( \rightarrow \) expression 2 (if 1 then 2)
Distribution:

```plaintext
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.
TEMPOERAL 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
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:

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

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.

$ represents a non-zero and finite number
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:
\[ r \#\#2 \ s \]
Initial Delay:

\[
n \text{s}
\]

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

Example: \#\#3 s

Range:

\[
r \#\#[M : N] \text{s}
\]

means that if \( r \) is true on current tick, \( s \) will be true \( M \) to \( N \) ticks from current tick.
Example:

\[ a \#\# 2 b \#\# 1 a \#\# 2 b \#\# 1 a \#\# 2 b \#\# 1 a \#\# 2 b \]

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
Go to Repetition:

\[ r \quad [\#\#] \quad 1 \quad s \quad [\rightarrow N] \quad [\#\#] \quad 1 \quad t \]

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

\[ r \quad [\#\#] \quad 1 \quad s \quad [\rightarrow M : N] \quad [\#\#] \quad 1 \quad t \]

Means \( r \) followed by at least \( M \), at most \( N \) \( s \)'s followed next tick by \( t \)

Example: \( e \quad [\rightarrow 2] \)
What does the following sequence mean?
\[a \#\#1 \ b \ [\rightarrow 2:10] \ #\#1 \ c\]

Watch out for the number of threads!

How can we interpret the following sequence?
\[a \ #\#1 \ b \ [\rightarrow 2:10] \ #\#1 \ c\]
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.
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:

\[( t e_1 \quad \#\#2 \quad t e_2 ) \quad \textbf{and} \quad ( t e_3 \quad \#\#2 \quad t e_4 \quad \#\#2 \quad t e_5 )\]

What if the two operands are Boolean expressions? 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.

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

Example:

\[( te1 \#\#[1:5] \ te2 ) \ intersect \ ( te3 \#\#2 \ te4 \#\#2 \ te5 ) \]
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.
How can we describe the following condition?

\(!trdy[*7] \text{ within } (\text{fell}(irdy) \#\# 1 \text{ !irdy}[*8])\)
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:

```verilog
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:

```verilog
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

sequence lfps_zero_detection_c;
  @(posedge clk)(
    (!oP_txelecidleAux) [*LOW_DURATION_1:LOW_DURATION_2] ##1
    (oP_txelecidleAux) [*HIGH_DURATION_1:HIGHS_DURATION_2] ##1
    (!oP_txelecidleAux)
  );
Endsequence : lfps_zero_detection_c
USB3.1 LFPS ONE DETECTION

sequence lfps_one_detection_c;
  @(posedge clk)(
    (!oP_txelecidleAux) [*LOW_DURATION_1:LOW_DURATION_2] ###1
    (oP_txelecidleAux) [*HIGH_DURATION_1:HIGHER_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
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 lead 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:

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

41
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:

```verbatim
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:

```verbatim
property prop_not;
    @(posedge clk) not property_expr;
endproperty : prop_not
```
**Property Type: Disjunction**

A property is a disjunction if it has the form:

\[
\text{property}\_\text{expr1} \text{ or property}\_\text{expr2}
\]

The property evaluates to true if, and only if, at least one of \text{property}\_\text{expr1} and \text{property}\_\text{expr2} evaluates to true.

**Property Type: Conjunction**

A property is a conjunction if it has the form:

\[
\text{property}\_\text{expr1} \text{ and property}\_\text{expr2}
\]

The property evaluates to true if, and only if, both \text{property}\_\text{expr1} and \text{property}\_\text{expr2} evaluate to true.
**Property Type: If ... Else**

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

```markdown
if (expression_or_dist) property_expr1
```

or the form

```markdown
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.

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

$$\text{sequence_expr} \mid-> \text{property_expr}$$

or the form (temporal)

$$\text{sequence_expr} \mid=\to \text{property_expr}$$
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:

```verilog
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.
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
- It may not contain temporal expressions
- May be inserted anywhere in the procedural code
- Evaluated as statement

Concurrent
- 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:

**default_usb_check:**

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

```vhdl
my_concurrent_check:

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

- **Assertion name (Label):** `my_concurrent_check`
- **Sampling event:** `posedge clk`
- **Asynchronous reset:** `rst`
- **Assertion body:**
  ```vhdl
  (disable iff (rst) not (a ##1 b))
  ```
- **Pass statement:** `$info$ ("Property p passed")`
- **Fail statement:** `$error$ ("Property p failed")`
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]
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.
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." );
```
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.

### Assume

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

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

**Specifies requirements for the environment.**
The assume statement follows this syntax:

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

See full description [1, A.2.10]

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:

```plaintext
property pr1;
    @( posedge clk )
    !reset_n |-> !req;  // when reset_n is asserted (0), keep req 0
endproperty
```
The following properties are assumed:

```plaintext
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] ##0 ack;
endproperty

assume_req1: assume property (pr1);
assume req2: assume property (pr2);
assume_req3: assume property (pr3);
```
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

<table>
<thead>
<tr>
<th>Description</th>
<th>Gain Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just comparing code to represent the sequence model:</td>
<td>~3.5x</td>
</tr>
<tr>
<td>Considering the built of coverage, assertion and the sequence encapsulation:</td>
<td>~1.7x</td>
</tr>
</tbody>
</table>

### Formal Temporal Logic vs Structural Modeling (Scoreboard)

<table>
<thead>
<tr>
<th>Description</th>
<th>Gain Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just comparing code to represent the sequence model:</td>
<td>~6x</td>
</tr>
<tr>
<td>Considering the built of coverage, assertion and the sequence encapsulation:</td>
<td>~3x</td>
</tr>
</tbody>
</table>

**Figure of Merit: Number of Code Lines**

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

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

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

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