Simulation Control

- Successful simulation of a testcase to completion involves the execution of the following major functions
  - Generating the testcase configuration
  - Building the verification environment around the DUT according to the generated testcase configuration
  - Disabling all assertions and resetting the DUT
  - Configuring the DUT according to the generated testcase configuration
  - Starting all transactors and generators in the environment
  - Detecting the end-of-test conditions
  - Stopping all generators in an orderly fashion
  - Draining the DUT and collecting statistics
  - Reporting on the success or failure of the simulation run

Data and Transactions

- Make effective use of the object-oriented programming model
- A data item shall be modeled using a class - Any atomic amount of data eventually or directly processed by the DUT
  - Packets, instructions, pixels, picture frames, SDH frames and ATM cells are all examples of data items.
  - A data item can be composed of smaller data items by composing a class from smaller classes.
  - For example, a class modeling a picture frame could be composed of thousands of instances of a class modeling individual pixels.
- Using the *class* construct has advantages over using *struct* or *union* constructs.
  - The latter would only be able to model the values contained in the data item; whereas, classes can also model operations and transformations on these data items using *methods*.
  - *Class* instances are more efficient to process and move around as only a reference to the instance is assigned or copied.
  - *Struct* and *union* instances are scalar variables and their entire content is always assigned or copied
- A *class* can also contain constraint declarations to control the randomization of data item values; whereas a *struct* and *union* cannot.
- Finally, it will be possible to modify the default behavior and constraints of a *class* through *inheritance*, without actually modifying the original base model.
- *Struct* and *union* do not support inheritance.

Data and Transactions

- This approach offers several advantages
- It is easy to create a series of random transactions. Generating random transactions becomes a process identical to generating random data (*rand* attribute)
- Random transactions can be constrained.
- New transactions can be added without modifying interfaces.
- It allows easier integration with the scoreboard.
  - Since a transaction is fully described as an object, a simple reference to that object instance passed to the scoreboard is enough to completely define the stimulus and derive the expected response.

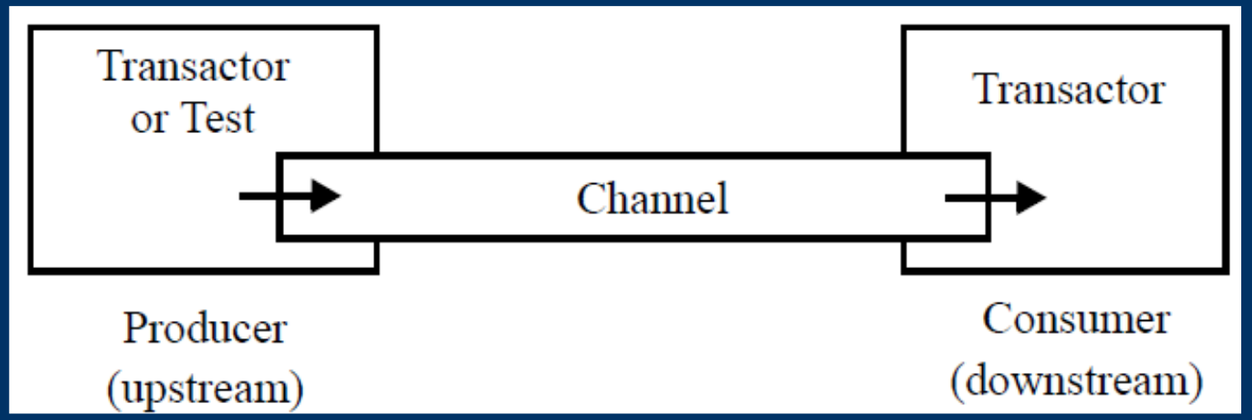
Transactors

- The term transactor is used to identify components of the verification environment that interface between two levels of abstractions for a particular protocol or to generate protocol transactions.
- Driver, Monitor, Checker and Generator are all transactors.
- The lifetime of transactors is static to the verification environment: They are created at the beginning of the simulation and stay in existence for the entire duration.
- They are structural components of the verification components; they are similar to modules in the DUT

Transaction-Level Interfaces

- The transaction-level interface lets the higher layers of the verification environment stimulate the DUT on a DUT interface.
- Transaction-level interfaces remove the higher-level layers from the physical interface details.
- In functional-layer transactors, a high-level transaction interface is used to receive a description of high-level transactions to be executed.
  - These high level transactions are executed by executing one or more lower level transactions
    - These lower-level transactions are submitted to a lower level transactor—in a lower functional sub-layer or in the command layer via a low-level transaction interface.
- Transaction-level interfaces are mechanisms to exchange transactions between two transactors or a directed testcase and a transactor.

Transaction-Level Interfaces



Transaction Interface Channel

Timing Interface

- Higher-layer transactors may require timing-related information as soon as that information is available, asynchronously from any transaction completion it may be associated with.

Callback Methods

- The behavior of a transactor has to be controllable as required by the verification environment and individual testcases without requiring modifications of the transactor itself.
- For example, callback methods can be used to monitor the data flowing through a transactor to check for correctness, inject errors or collect functional coverage metrics.

Ad-Hoc Testbenches

- Ad-hoc testbenches are quickly put together, typically by design engineers, to create some simple stimulus to verify the basic operation of a design unit.
- The response is usually checked visually.

Legacy Bus-Functional Model

- Bus Functional Model (BFM) for a device interacts with the DUT by both driving and sampling the DUT signals.
- A bus functional model is a model that provides a task or procedural interface to specify certain bus operations for a defined bus protocol.
- For a memory DUT, transactions usually take the form of read and write operations.
- It has to follow the timing protocol of the DUT interface.
- BFM describes the functionality and provides a cycle accurate interface to DUT.
- It models external behavior of the device.

STIMULUS AND RESPONSE

- Generating Stimulus
- Controlling Random Generation
- Self-Checking Structures

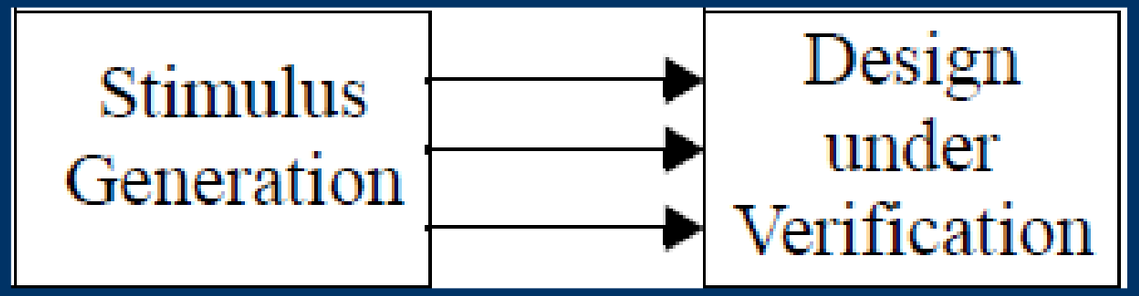
Reference: Writing Testbenches using System Verilog by JanickvBergeron

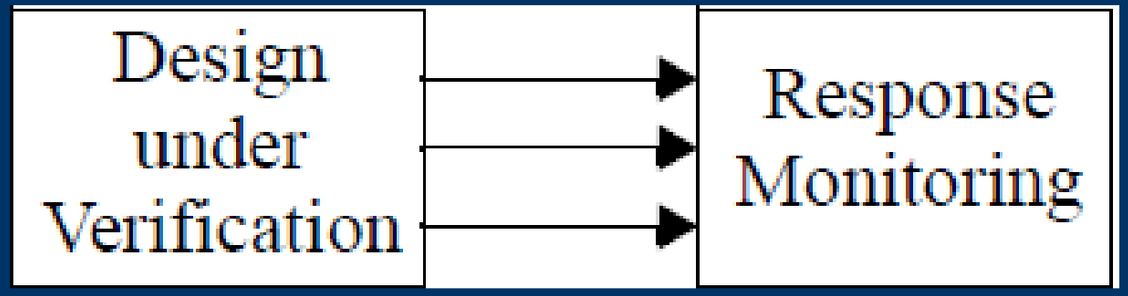
STIMULUS AND RESPONSE

- Generating Stimulus
  - Reference Signals
  - Time Resolution Issues
  - Applying Reset

Generating Stimulus

- The purpose of writing testbenches is to apply stimulus to a design and observe the response.
- That response must then be compared against the expected behavior.
- Generating stimulus is the process of providing input signals to the design under verification
- Monitoring is the process of observing output signals from the design under verification





Generating Stimulus

- Reference Signals - clock signal has a very simple repetitive pattern, it is one of the first and most fundamental signals to generate

```
module tb_top;
bit clk = 0;
always #5 clk = ~clk;
...
endmodule
```

Apparently complex waveform

```
always
begin
    S = 1'b0; #20ns;
    S = 1'b1; #10ns;
    S = 1'b0; #10ns;
    S = 1'b1; #20ns;
    S = 1'b0; #50ns;
    S = 1'b1; #10ns;
    S = 1'b0; #20ns;
    S = 1'b1; #10ns;
    S = 1'b0; #20ns;
    S = 1'b1; #40ns;
    S = 1'b0; #20ns;
    ...
end
```

Generating a deterministic waveform

Generating Stimulus

- Time Resolution Issues - select the appropriate timescale and precision
- Make sure that integer operations do not truncate a fractional part

```
`timescale 1ns/1ns
module testbench;
...
bit clk = 0;
parameter cycle = 15;
always
begin
   #(cycle/2);  // Integer division
   clk =~clk;
end
endmodule
```

Truncation errors in stimulus generation

```
`timescale 1ns/1ns
...
#(cycle/2.0);   // Real division
```

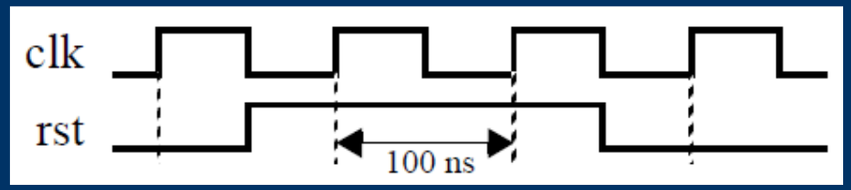
Rounding errors in stimulus generation

```
`timescale 1ns/100ps
...
#(cycle/2.0);   // Real division
```

Proper precision in stimulus generation

Generating Stimulus

- Applying Reset
  - The first signal to be generated after the clock signals is the hardware reset signal
  - The reset signal must be shaped properly to reset the design correctly
  - The generation of a synchronous reset signal should also reflect its synchronization with any clock signal



Reset waveform specification

```
bit clk = 0;
always #50 clk = ~clk;

bit rst = 0;
initial
begin
   #50  rst = 1'b1;
   #200 rst = 1'b0;
end
```

Possible race condition

```
bit clk = 0;
always #50 clk <= ~clk;

bit rst = 0;
initial
begin
   #50  rst <= 1'b1;
   #200 rst <= 1'b0;
end
```

Race-free generation

```
bit clk = 0;
always #50 clk = ~clk;

bit rst = 0;
initial
begin
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
end
```

Proper generation

Generating Stimulus

- One may generate reset from within a module task
- But reset task should be in program block

```
module tb_top;

bit clk = 0;
bit rst = 0;
always #50 clk = ~clk;
...
task hw_reset
   rst = 1'b0;
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
endtask: hw_reset
initial hw_reset;
...
endmodule
```

```
program test;

task hw_reset
   tb_top.rst <= 1'b0;
   @ (negedge tb_top.clk);
   tb_top.rst <= 1'b1;
   @ (negedge tb_top.clk);
   @ (negedge tb_top.clk);
   tb_top.rst <= 1'b0;
endtask: hw_reset

initial
begin
   hw_reset;
   ...
end
endprogram
```

Controlling Random Generation

```
bit clk100 = 0;
initial #({$random} % 10)
   forever #5 clk100 = ~clk100;
```

- Random Generation of Reference Signal Parameters
- In Verilog only system functions like $random are used for stimulus generation.
- SystemVerilog has randomization constructs
  - Constraints : Random variable generated in Boolean expressions, foreach, set membership, inline constraints, rand case, rand sequence, Conditional constraints and implication constraints.
  - Randomization : random function, constrained and unconstrained randomization, uniform distribution, weighted distribution, weighted range, weighted case, pre randomization, post randomization, declaration of random variable and non repeating sequence.
  - Dynamic constraints : disabling/enabling constraints, disabling/enabling random variables and overriding of constraint blocks.

Self-Checking Structures

- Automate output verification is to include the expected output with the input stimulus for every clock cycle
- Test vectors require synchronous interfaces
- Golden vectors - A set of reference simulation results can be used.
  - kept in a file or database.
- Golden vectors have an advantage over input/output vectors because the expected output values need not be specified in advance.
- If the simulation results are kept in ASCII files, the simplest comparison process involves using the UNIX *diff* utility.

Coverage-Driven Verification

- Coverage Metrics
- Coverage Models
- Functional Coverage Implementation
- Feedback Mechanisms

- Reference: Chapter 6, VMM, Janick Bergeron

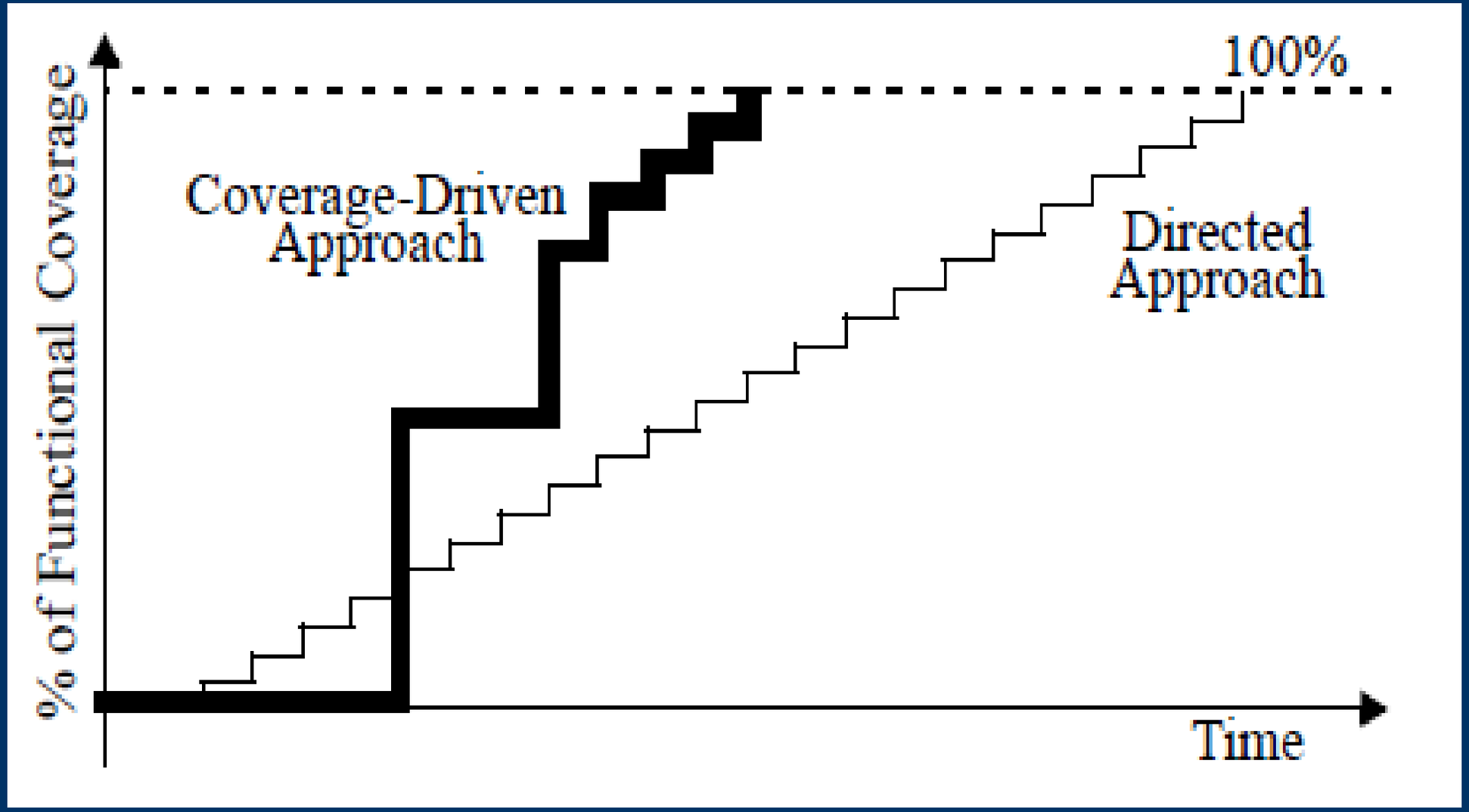
Coverage-Driven Verification

- Traditionally, coverage is used as a confidence-building metric.
- Used as a safety net to ensure that the verification plan was as complete and that the design was verified as thoroughly as possible.
- Coverage measurements are done toward the end of the verification process, when the bulk of the testcases have been written.
- The coverage metrics are usually expected to be initially relatively high
  - Tweak the test suite to increase the coverage numbers
  - Or, justify why some of these numbers cannot—or need not—reach 100%.

Coverage-Driven Verification

- Coverage-driven Random-based Approach
  - Random verification does not mean that apply zeroes and ones to every input signal in the design.
  - It is the sequence and timing of these operations and the content of the data transferred that is random.
  - Through the addition of constraints, a random testbench can be steered toward exercising specific features.
  - Measure progress against functional coverage points that will identify whether a feature has been exercised.
  - The objective becomes filling a functional coverage model of the design rather than writing a series of testcases.
  - Fill the coverage model using large directed testbenches or let a random testbench create the testcases and exercise the features for you

Coverage-Driven Verification



Coverage-Driven Verification

Directed Verification

- Test write have to list out all features to be verified
- Difficult find the corner case bugs.
- Stimulus is generated in the test case
- Has limited amount of randomization.
- Becomes tedious when design complexity increases
- Test case maintenance will be harder for large designs

Constraint-Driven Random Verification

- Test writer has to specify the set of constraints.
- Easy to find the corner case bugs
- Stimulus is generated in testbench depending on the given constraints.
- Number of testcases can be reduced.
- Testcase maintenance is easy.
- Dis-adv: Testbench may generate similar scenario many times.

Coverage Metrics

- Coverage metrics are measures of collected coverage data against stated or implied goals, usually expressed as percentage.
- Coverage data is collected in one or more databases by simulation or static tools.
- Coverage analysis or reporting tools compare the collected data, usually aggregated from multiple simulations or static analysis, against the goal for each coverage point.
- If the collected data fully covers the goal, the coverage metric is reported as 100%.

Coverage Models

- A coverage model is a definition of the stimulus/response space subset that will demonstrate, with an acceptable degree of confidence, that the functionality of the design is correct
- For example, the solution space for a 32-bit adder is a three-dimensional space measuring 2^{32} x 2^{32} x 2 (input A x input B x carry-in).
- Exhaustively verifying this simple design, assuming one set of input values can be verified every nanosecond, and would require over one thousand years.
- But an adequate level of confidence in the correctness of an implementation can be obtained by using just a few sets of input pairs (e.g., all possible combinations of walking-ones, walking-zeroes, all-ones and all-zeroes—less than one thousand input patterns).
- That set of inputs is the coverage model for the combinatorial adder.

Coverage Models

- A coverage model is composed of *structural coverage* and *functional coverage* target definitions.
- These definitions can be further refined into sub-metrics like FSM coverage, expression coverage, cross-coverage and assertion coverage.
- Coverage metrics from simulation, formal analysis and structural analysis tools are combined for an overall assessment of verification progress.

Coverage Models

- **Structural Coverage Modeling**
- Models are implicitly defined by the code used to implement the design.
- Includes *line* coverage, *expression* coverage, *toggle* coverage and *FSM* coverage, *assertion coverage*.
- The collection of structural coverage metrics is automatically inserted and enabled by tools.
  - Analysis requires action by the engineers.
- Limitations
  - It can only be used to measure how thoroughly an existing implementation has been exercised.
  - It cannot identify functionality that has not yet been implemented neither can it determine if the intent of a testcase has been achieved.

Coverage Models

- **Functional Coverage Modeling**
- A verification plan typically details the individual testcases that must be written to verify a particular design.
- The plan eventually is summarized into a table, with the testcase name in a column, the name of the verification engineer assigned to that testcase in another column and an indication of the completion status of the testcase in another.
- It may be implemented in a directed testcase, hit by pure chance by a random simulation or proven to be correct by a formal analysis.
- In a coverage-driven verification process, the strategy becomes how to use the best implementation vehicle to hit the most coverage points with the least effort.
- If individual directed testbenches are required to hit individual coverage points, it will be necessary to write as many testbenches as there are coverage points.
- By adding randomness and simulating the same testbench with several different seeds, the same testbench may be able to hit multiple coverage points, effectively replacing multiple directed testbenches.

Coverage Models

- **Functional Coverage Modeling**
- Like structural coverage, functional coverage has limitations.
- It can only demonstrate that some interesting condition has been reached, not that it was reached correctly or that the corresponding responses were correct.
- It does not imply completeness: If a functional coverage models only 70 percent of the DUT’s functionality, then reaching 100 percent functional coverage will only verify that 70 percent.
- Functional coverage is an expression of the verification requirements and intent. Therefore, it cannot be automatically derived or implemented from the DUT or testbench source code.
- There is no way to ensure that a functional coverage model is complete because it is only a reflection of the verification plan, and there is no automated way to ensure that the verification plan itself is complete.

Coverage Models

- **Functional Coverage Modeling**
- A functional coverage model should be subjected to the same type of development and review process as the verification plan to ensure its completeness.
- It will also evolve throughout the duration of the verification project, as new functionality is added or new corner cases are identified.

FUNCTIONAL COVERAGE IMPLEMENTATION

- Metrics in a structural coverage model can be automatically generated and collected.
- For example, to obtain a rating of the line execution coverage, all that is required is the specification of a command-line option.
- No additional work is required by the user
- But a functional coverage model must be manually specified and captured.
- Functional coverage—which models the functional verification requirements—captures the intent of the verification effort.
- Because intent cannot be automatically derived from the implementation, functional coverage must be independently captured.
- Two constructs are available in SystemVerilog to specify functional coverage: coverage groups and coverage properties.

FUNCTIONAL COVERAGE IMPLEMENTATION

```
covergroup cg(ref int array, int low, int high ) @(clk);
  coverpoint// sample variable passed by reference
  {
    bins s = { [low : high] };
  }
endgroup

int A, B;
rgc1 = new( A, 0, 50 );// cover A in range 0 to 50
rgc2 = new( B, 120, 600 );// cover B in range 120 to 600
```

- A coverage group, cg, in which the signal to be sampled as well as the extent of the coverage bins are specified as arguments.
- Two instances of the coverage group are created; each instance samples a different signal and covers a different range of values.

FUNCTIONAL COVERAGE IMPLEMENTATION

```
sequence s;
    @(posedge clk) a ##1 b;
  endsequence

  property p;
    a |-> s;
  endproperty

  assert property (p);
  cover property (p);
```

- 'assert' will check a failure of the rule/property, 'cover' will check whether the rule/property was ever fulfilled
- The difference is that covering a property ignores the failures, and asserting a property ignores the passes.

Feedback Mechanisms

- In a coverage-driven verification process, the analysis of the collected coverage data is used to identify where to focus the next verification efforts.
- Thus, a feedback mechanism is required to translate uncovered areas in the coverage model into additional stimulus or analysis.
- **Recommendation**
- More simulations with different seeds should be run if the input coverage is low and few simulations have been run
- A new test should be created if constraints need to be modified or new scenarios defined.
- A directed test should be used if a scenario is too complex to define or unlikely to happen randomly
- Formal technology should be used if the problem size fits the capacity of the tool and a coverage property is used

Assertion Based Verification

Reference:
VMM by Janick Bergen

Assertion

- Specifying Assertions
- Assertions on Internal DUT Signals
- Assertions on External Interfaces
- Assertion Coding Guidelines
- Reusable Assertion-Based Checkers
- Qualification of Assertions

What is an Assertion?

- An assertion is a description of a property of the design
  - If a property that is being checked does not behave the way it is expected, the assertion fails.
  - If a property that is forbidden from happening in a design happens, the assertion fails.
- Assertions can be continuously monitored during functional simulations.
- The same assertions can also be re-used for verifying the design using formal techniques.
- Assertions, also known as monitors or checker.

What is an Assertion?

- Assertions are primarily used to validate the behaviour of a design. ("Is it working correctly?")
- They may also be used to provide functional coverage information for a design ("How good is the test?").
- Assertions can be checked dynamically by simulation, or statically by a separate property checker tool – i.e. a formal verification tool that proves whether or not a design meets its specification.
- Such tools may require certain assumptions about the design's behavior to be specified.

Example : After the posedge of (a), (b) should also rise in 1 to 3 clock cycles

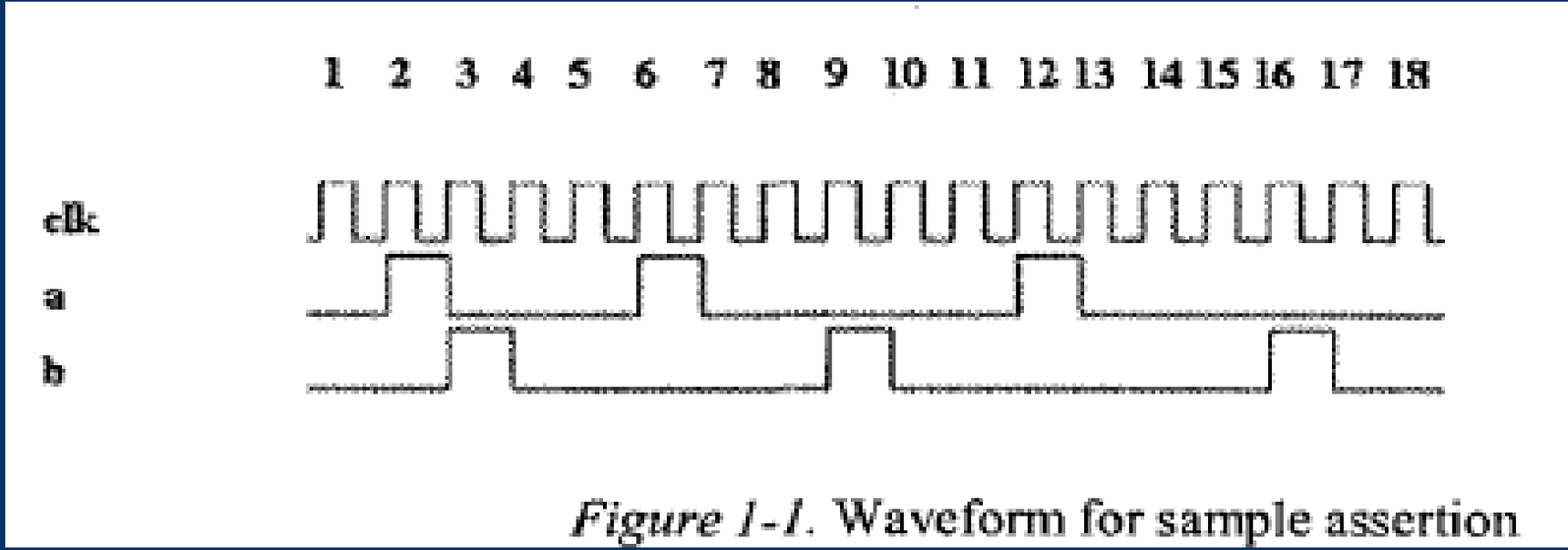


Figure 1-1. Waveform for sample assertion

Verilog code to check posedge of b

```
repeat (1) @(posedge clk);
  fork: a_to_b

     begin
     @(posedge b)
     $display
     ("SUCCESS: b arrived in time\n", $time);
     disable a_to_b;
     end

     begin
     repeat (3) @(posedge clk);
     $display
     ("ERROR:b did not arrive in time\n", $time);
     disable a_to_b;
     end

   join
```

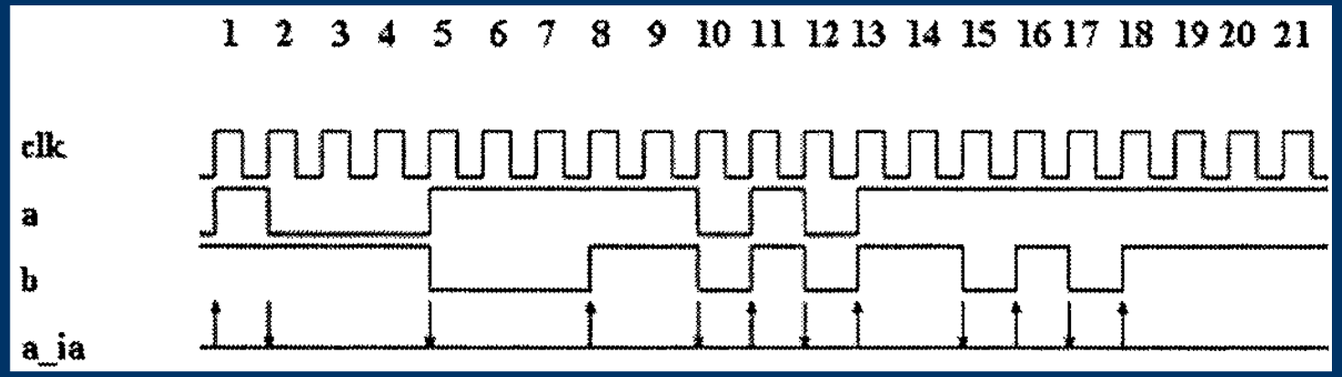
SystemVerilog code to check posedge of b

```
a_to_b_chk:
assert property
@(posedge clk) $rose(a) |-> ##[1:3] $rose(b));
```

SVA Terminology

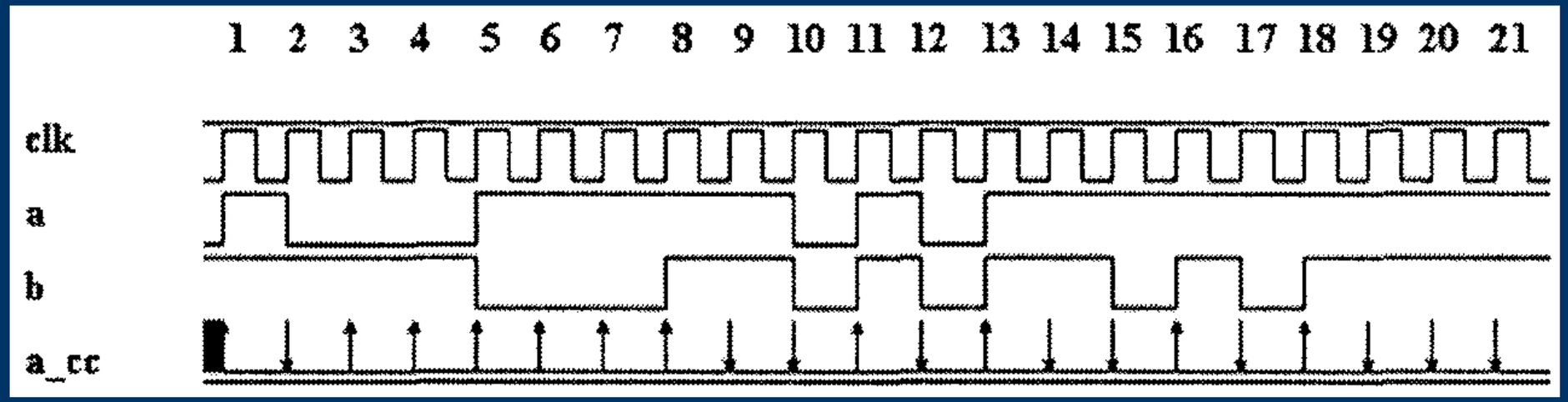
- Immediate assertions and Concurrent assertions
- **Immediate assertions**
  - Based on simulation event semantics.
  - Test expression is evaluated just like any other Verilog expression within a procedural block.
  - These are not temporal in nature and are evaluated immediately.
  - Have to be placed in a procedural block definition.
  - Used only with dynamic simulation.

```
always_comb
begin
    a_ia: assert (a && b);
end
```



SVA Terminology

```
a_cc: assert property(@(posedge clk)
                      not (a && b));
```



- The key concept in this example is that the property is being verified on every positive edge of the clock irrespective of whether or not signal "a" and signal "b" changes.

SVA Terminology

- Immediate assertions
- Procedural statements and are mainly used in simulation.
- An assertion is basically a statement that something must be true, similar to the if statement.
- The difference is that an if statement does not assert that an expression is true, it simply checks that it is true

```
if (A == B) ... // Simply checks if A equals B
assert (A == B); // Asserts that A equals B; if not, an error is generated
```

```
assert (A == B) $display ("OK. A equals B");
```

```
 assert (A == B) else $error("It's gone wrong");
```

SVA Terminology

- Immediate assertions
- Three severity system tasks that can be included in the fail statement to specify a severity level: $fatal, $error (the default severity) and $warning.
- In addition, the system task $info indicates that the assertion failure carries no specific severity.

```
ReadCheck: assert (data === correct_data)
             else $error("memory read error");
    Igt10: assert (I > 10)
             else $warning("I is less than or equal to 10");
```

```
AeqB: assert (a === b)
      else begin
        error_count++;
        $error("A should equal B");
      end
```

SVA Terminology

- Concurrent assertions and Immediate assertions
- **Concurrent assertions**
  - Based on clock cycles.
  - Test expression is evaluated at clock edges based on the sampled values of the variables involved.
  - Sampling of variables is done in the "preponed" region and the evaluation of the expression is done in the "observed" region of the scheduler.
  - Can be placed in a procedural block, a module, an interface or a program definition.
  - Can be used with both static (formal) and dynamic verification (simulation) tools.

SVA Terminology

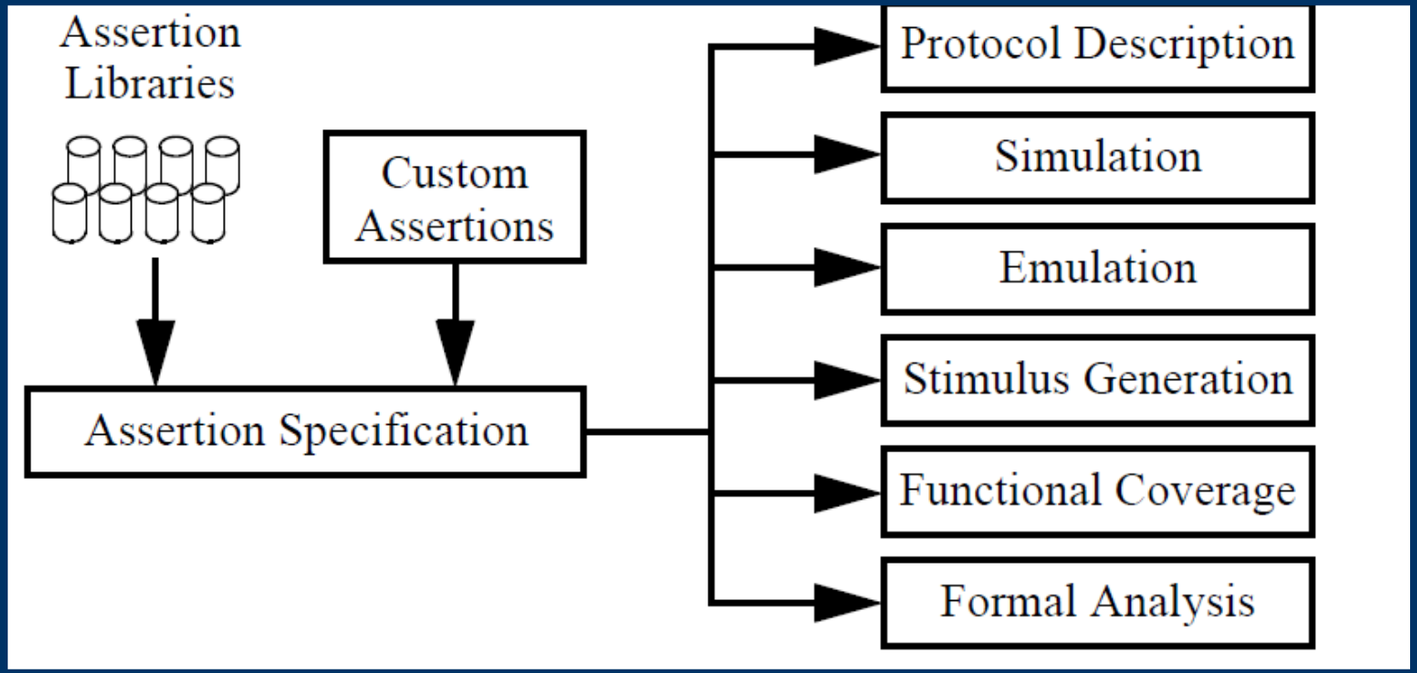
- Concurrent assertions are used to check behavior such as
- "The Read and Write signals should never be asserted together."
- "A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."

```
assert property (!(Read && Write));//checked at every point in the simulation
```

```
assert property (@(posedge Clock)
  Req |-> ##[1:2] Ack);// only checked when a rising clock edge has occurred
```

Specifying Assertions

- Assertions are observers that monitor signals in the DUT for correct behavior.
- In SystemVerilog, assertions are expressed using a declarative language.



Assertions in a Verification Process

Specifying Assertions

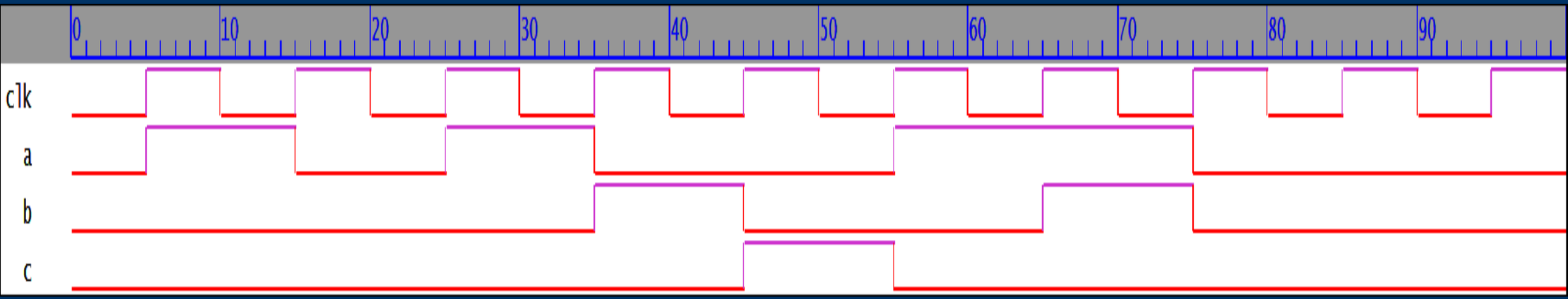
- Concurrent assertions
  - Embedded in functional code and ignored in synthesis
- Example
  - read_writeCheck: assert property ((@posedge clk) !(wr_enable && rd_enable))
- Naming and asserting the property
  1. Define the property inline with assertion statement
     - My_Assert: assert property (@(negedge clk) (a &&b))
  2. Name and Assert as separate statements
     - property My_Property:
     - (@(negedge clk) (a &&b))
     - endproperty

     - My_Assert: assert property (My_ Property);

Specifying Assertions

- Concurrent assertions
- Sequences
  - @(posedge clk) P ##1 Q ##1 R
- Same cycle sequence implication using | ->
  - property my_prop;
  - @(posedge clk)
  - (req && !ack) |-> busy;
  - endproperty
- Next cycle sequence implication using | =>
  - property my_prop;
  - @(posedge clk)
  - (req && !ack) |=> busy;
  - endproperty

Specifying Assertions




```
property state;
  @(posedge clk)
  (a ##1 b) |=> c;
endproperty

My_Assert: assert property (state);
```

Output

Assertion "My_Assert" FAILED at time: 85ns (9 clk), scope: test_example_assertion, start-time: 65ns (7 clk)

Specifying Assertions

```
module example_assertion (input bit clk, output bit a, b, c);
  initial begin
    @(posedge clk) begin
      a = 1;
      b = 0;
      c = 0;
    end
    @(posedge clk) begin
      a = 0;
    end
    @(posedge clk) begin
      a = 1;
    end
    @(posedge clk) begin
      a = 0;
      b = 1;
     end
    @(posedge clk) begin
      b = 0;
      c = 1;
     end
    @(posedge clk) begin
      a = 1;
      c = 0;
    end
    @(posedge clk) begin
      b = 1;
      c = 0;
    end
    @(posedge clk) begin
      a = 0;
      b = 0;
    end
  end

endmodule
```

```
module test_example_assertion;
  bit clk,a,b,c;

  example_assertion duv(.*);

  always
    #5 clk = ~clk;

  property state;
    @(posedge clk)
    (a ##1 b) |=> c;
   // a ##1 b ##1 c;
  endproperty

  My_Assert: assert property (state);


  initial begin
    $dumpfile("my_asertion.vcd");
    $dumpvars;
    #100 $finish;
  end

endmodule
```

Specifying Assertions

```
property d_ff;
  @(posedge clk) disable iff(rst)
  (##1 out ==$past(d));
endproperty

property chk_rst;
  @(rst)
    rst |-> out == 0;
endproperty

My_property1: assert property (d_ff);
My_property2: assert property (chk_rst);
```

Try: D Flip flop with asynchrnous reset

Specifying Assertions

- Cycle delay repetition
  - property ab;
  - @(posedge clk)
  - !a |=> ## 4 b;
  - endproperty
- Consecutive sequence repetition
  - property ab;
  - @(posedge clk)
  - !a[*5] |=> a;
  - endproperty

Assertions on Internal DUT Signals

- Assertions and coverage points on internal signals are inserted by the designer during the detailed implementation of a block
- Inlined Concurrent Assertion
  - Within a block, like the initial, always, always_comb, always_ff, final, task, function

```
module moduleA(input bit clk, output ...);
    ... // design code
    check_progress : assert property (
    @(posedge clk) disable iff (reset)
    (valid_in && !stall) ##1 (!stall[->3]) |-> ##1 f3 );
    ... // design code
endmodule : moduleA
```

Assertions on External Interfaces

- Assertions on external interfaces are used by verification engineers.
- The assertions can be created by them or can already be provided by designers on block interfaces that become exposed as external system interfaces.
- Assertions on external interfaces are derived from the functional specification of the interface signals and protocols and coverage statements are related to the verification plan
- The DUT is generally viewed as a black box
- Assertions shall be divided into two categories: assertions on local interface protocols and assertions on signals from two or more interfaces.

Assertions Coding Guidelines

Reusable Assertion-Based Checkers

- Reusable checkers can verify some relatively simple properties typically found in any design.
- They are usually included in a library of checkers.
- The VMM checker library is an example of simple reusable checkers.
- Assertion IP can verify complete behaviors of specific standard protocols and behaviors, such as PCI, SPI, AMBA Protocol Family
- Checkers shall be packaged using an interface
  - interfaces can be instantiated in either a module, an interface or a program.
- The inclusion of assert property statements in the checker shall be controlled by the macro ASSERT_ON.
- The inclusion of cover property statements in the checker shall be controlled by the macro COVER_ON.

Qualification of Assertions

- Assertions have to be thoroughly verified for compliance with their requirements.
- Bugs can arise due to improper implementation of the assertion itself, misinterpretation of the assertion language constructs, incorrect parameter values and port connections, typographical errors, or ambiguity in the protocol specification.
- An error in an assertion or in the deployment of a checker may result in false reporting of successful design verification.
- The qualification problem is complex because, not only the acceptance of correct protocol behavior has to be verified, but also that violations of the specification must be detected and reported by the assertion or checker.
- In other words, both the good and the wrong behaviors must be thoroughly verified.
- Different ways to verify that a given assertion or checker complies with the requirements.
  - Hybrid formal tools may be used

Qualification of Assertions

- **Visual inspection —** This inspection may be sufficient if the abstraction level of the assertion matches the natural language specification.
- Visual inspection requires solid understanding of assertion language semantics.
- Typically, this is used on simple assertions involving a few temporal operators.
- For example,
  - c |-> d[*1:$] ##1 e;
  - It is sufficient to make sure that the Boolean expressions *c*, *d* and *e* are correct., and that the timing cases are valid.

- **In-situ —** Inline with the DUT, is the easiest method to implement, but may only exercise a subset of all accepting sequences.

- **Assertion testbenches —** A testbench can be written to exercise the assertion or the reusable checker, like any other DUT.

- **Formal verification —** The formal tool is used to generate random test sequences that satisfy the assertions as assumptions and observe them in a waveform viewer to check their validity.