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:

```
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

- 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

Simple Logic Expressions

Sequences

Implication of Sequences

Properties

Assertion Statements

Action!
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.
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).

**Definition**

- and
- or
- Non-temporal implication:
  
  expression 1 \(\rightarrow\) expression 2 (if 1 then 2)
Distribution:

expression \texttt{dist} \{ \texttt{dist\_list} \};

dist\_list \ := \ \texttt{dist\_item} \{, \texttt{dist\_item}\}

dist\_item \ := \ \texttt{value\_range} [ \texttt{dist\_weight} ]

dist\_weight \ := \ (\,:= \texttt{expression}) \mid (\,:/ \texttt{expression})

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

Example:

\texttt{usb\_symbol dist \{100 := 1, 200 := 2, 300 := 5\}}

It means \texttt{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
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```
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:

\[
\text{## integral_number or identifier} \\
\text{## ( constant_expression )} \\
\text{## [ cycle_delay_const_range_expression ]}
\]

cycle_delay_const_range_expression := const:const or const:

Example:

\[
\text{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 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.
Example:

\[
\begin{align*}
& a \#\# 2 \ b \ #\# 1 \ a \ #\# 2 \ b \ #\# 1 \ a \ #\# 2 \ b \ #\# 1 \ a \ #\# 2 \ b \\
\end{align*}
\]

by simplification the previous sequence results in:

\[
( a \ #\# 2 \ b ) [*5]
\]

\[
\begin{align*}
& r \ #\#[*M : N] s \\
\end{align*}
\]

Repeat \( r \) at least \( M \) times and as many as \( N \) times consecutively

\[
\begin{align*}
& r \ #\#[ *M : $ \] \\
\end{align*}
\]

Repeat \( r \) an unknown number of times but at least \( M \) times
Go to Repetition:
\[ r \text{##} 1 \text{s [->} N \text{]} \text{##} 1 \text{t} \] 
Means \( r \) followed by exactly \( N \) not necessarily consecutive \( s \)'s with last \( s \) followed the next tick by \( t \)

\[ r \text{##} 1 \text{s [->} M:N \text{]} \text{##} 1 \text{t} \] 
Means \( r \) followed by at least \( M \), at most \( N \) \( s \)'s followed next tick by \( t \)

Example: \( e \text{[->2]} \)

Non-Consecutive Repetition
\[ r \text{##} 1 \text{s [=} N \text{]} \text{##} 1 \text{t} \] 
Means \( r \) followed by exactly \( N \) not necessarily consecutive \( s \)'s with last \( s \) followed sometime by \( t \)

\[ r \text{##} 1 \text{s [=} M:N \text{]} \text{##} 1 \text{t} \] 
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

How can we interpret the following sequence?

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

Watch out for the number of threads!
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:

\[(te1 \#\#2 te2) \textbf{and} (te3 \#\#2 te4 \#\#2 te5)\]

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:

\[
( \text{te1} \# [1:5] \text{te2} ) \text{ intersect } ( \text{te3} \# \# 2 \text{te4} \# \# 2 \text{te5} )
\]
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?

$!trdy[*7]$ 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:

```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_DUR:LOW_DUR] ##1
    (oP_txelecidleAux) [*HIGH_DUR:HIGH_DUR] ##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:HIG...
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
   );
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:

```
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:

```verilog
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:

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

A property is a disjunction if it has the form:

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

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:

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

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:

```plaintext
if (expression_or_dist) property_expr1
```

or the form

```plaintext
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 $|=>$. Therefore, a property is an implication if it has either the form (non-temporal)

$$\text{sequence_expr} \, |-> \, \text{property_expr}$$

or the form (temporal)

$$\text{sequence_expr} \, |=> \, \text{property_expr}$$
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 $\rightarrow$ 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
```

![Graph showing data transfer timeline]
V. ASSERTION LANGUAGE
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:
    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.
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:

my_concurrent_check:

assert property (@ (posedge clk) disable iff (rst) not (a ##1 b)) $info ("Property p passed"); else $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 ) )
$ddisplay( "env_prop passed." ); else $ddisplay( " 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:

```vhdl
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:

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

```
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
```

assumereq1: assume property (pr1);
assumereq2: assume property (pr2);
assumereq3: 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:

\[
\text{cover property ( property_spec ) statement_or_null}
\]

See full description [1, A.2.10]

Example:

\[
\text{cover property ( @ ( posedge clk ) !rst throughout req ##1 ack );}
\]
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

**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