A quick-start guide to PSL – Property Specification Language

1 Introduction

Property based verification is picking up momentum among the design and verification community. PSL has emerged as one of the standard assertion languages and is on its way to becoming an IEEE standard. This tutorial is intended to get the readers quickly started on the language. It does not however cover the depth and breadth of the language, history of the language, why ABV is required etc.

1.1 What is an assertion language?

An assertion or property language captures the design behavior spread across multiple clock cycles in a concise, unambiguous manner. It is a great way to describe control intensive design behaviors, pipelines, latencies etc. While traditional RTL captures the cycle-by-cycle behaviour, it is way too detailed (and error prone) to describe properties at a higher level. While the bottom most layer of a property is still a boolean expression, a property language adds means to express temporal relationships among those expressions and provides operators to capture complex design behaviors in a concise manner.

PSL – Property Specification Language is designed to capture design intent in an executable, formal, unambiguous manner. It is developed as more "evolutionary" language than re-inventing wheel. It uses many of the underlying HDL operators and expression syntax to build the boolean expressions in properties rather than defining its own syntax and semantics for the same. However, where-ever required, it defines its own syntax to build complex temporal relationship among the boolean expressions.

2 Anatomy of a PSL assertion

A quick start example:

Figure – 1, Anatomy of a PSL assertion

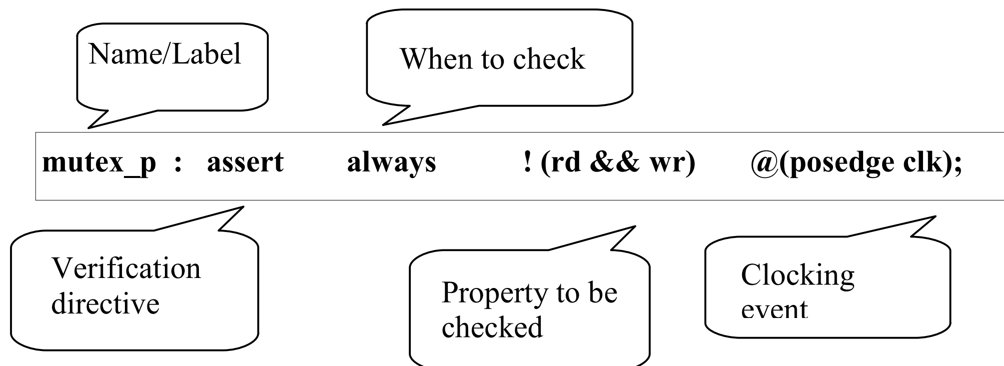

Figure 1 shows various pieces that comprise a complete PSL assertion. The next sections delve into individual pieces.

2.1 Label

PSL allows optional label to be specified for every assertion. It is a recommended practice to use a meaningful name to each assertion. It helps in identifying any failures, success reports coming from a PSL aware tool.

2.2 Verification directive

PSL provides a rich set of constructs to build complex properties. A property by definition is a declaration and a verification tool doesn't know what to do with it unless told otherwise. A Verification directive sits on top of a property and instructs a tool (simulator or a model checker) whether to "check" that the property is never violated, or "look for occurrence of the property" etc.

2.3 When to check?

As part of the temporal layer, PSL provides means to specify "when to check for a property". Loosely speaking they can also be called as "occurrence operators". PSL supports the following occurrence operators:

- always
- never
- eventually!
- next (family of them)

The "always" operator is the most frequently used one and it specifies that the following property expression should be checked every clock.

2.4 Property to be checked

This forms the core of PSL. PSL allows properties to be first declared and then used in an assertion or simply specify the complete property in the assertion itself as shown in Figure – 1.

2.5 Clocking event

Properties can be either clocked or unclocked. For now this tutorial focuses on clocked properties. A clock for a property can be either specified inline in the property definition itself as shown in Figure – 1. The symbol @ is used to specify clocking. Any boolean expression can be used as clock. PSL also allows a "default" clock specification, syntax of which is shown below.

```
// Using Verilog flavor.
default clock = (posedge clk);
```

Then a simple assertion can be specified like:

```
a_with_implicit_clk : assert always {fifo_full |-> !wr_fifo};
```

The above assertion will derive its clock from the default clock statement and is equivalent to the following:

```
a_with_explicit_clk : assert always {fifo_full |-> !wr_fifo} @(posedge clk);
```

2.6 Flavors, layers, examples

PSL is a multi-flavored, multi-layered language. This tutorial will walk through different layers in detail, but for the sake of simplicity will not focus on the different flavors. However the following few examples shall attempt in filling that gap.

The assertion example in Figure-1 is a Verilog flavored one, a similar one written in VHDL flavor will look like:

```
mutex_p : assert always not(rd and wr) @(rising_edge(clk));
```

As it can be seen above, the flavor dictates the syntax for the boolean expressions, clocking etc.

PSL is a well structured, layered language. It has four layers viz. boolean, temporal, verification and modeling layer. Each layer brings in a unique expressiveness to PSL. These layers are described in detail in later sections. A few quick start examples are shown below to demonstrate the concept of building PSL properties using individual layers. First a boolean expression is shown, as part of boolean layer. Temporal layer in PSL consists of SEREs/Sequences and properties. A simple sequence example shows how to build a sequence using boolean expressions as leaf elements. Then a property example is constructed using the previously defined sequences and combining them with Property operators. Finally, a verification directive (assert) is applied on the previously defined property to demonstrate the verification layer.

```
  -- BOOLEAN LAYER
  not (rd and wr); -- rd, wr are design signals.

-- TEMPORAL LAYER
-- Sequence definition
sequence s1 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_eop};

sequence s2 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_aborted};
-- In s1 & s2, the individual sequence elements (such as pkt_sop, pkt_xfer_en_n etc.) are
design signals.

-- Property definition
property p1 is reset_cycle_ended |=> {s1; s2};

-- Property p1 uses previously defined sequences s1 & s2.

-- VERIFICATION LAYER
a1 : assert p1;

-- Verification directive is being used on a previously defined property p1.
```

3 Property Expressions

A property expression can be as simple as a boolean equation spanning just one clock cycle or can be a complex temporal SERE (Sequential Extended Regular Expression) spanning multiple (and possibly variable number of) cycles.

3.1 Boolean layer

The lowest level of any PSL property is a boolean expression, and PSL avoids re-inventing wheel here. It derives its boolean expressive power entirely from the underlying HDL. There are currently 4 different flavors defined in 1.1 version of LRM viz. Verilog, VHDL, SystemVerilog and GDL flavors. Of these the GDL flavor is more of a place holder for a generic description layer and is not completely developed yet. Recently a SystemC flavor is being proposed and also a “portable PSL” that shall enable properties written in one flavor with a design described in a different flavor.

3.1.1 Verilog flavor

For Verilog, the following are few sample legal boolean expressions.

```
 mem_rd && mem_wr
!pkt_sop || (pkt_stop && pkt_eop)
```

3.1.2 VHDL flavor

An equivalent expression of the same in VHDL flavor is shown below.

mem_rd and mem_wr
(not pkt_sop) or (pkt_sop and pkt_eop)

Infact most of the boolean layer is available as part of VHDL's built-in assert construct.

As one can see, at the boolean layer, PSL looks quite similar to the underlying HDL and this is characteristic of PSL often referred to as "language neutrality".

3.2 Temporal Layer

While boolean layer forms the foundation of PSL, the real power of PSL comes from its temporal layer. The term "Temporal" refers to the design behavior expressed as a series of boolean expressions over multiple clock cycles. To support this, PSL has two major components in the temporal layer: Sequences and Properties. Sequences are built from basic boolean expressions and using sequence operators such as repetition operators. Properties are built on top of sequences and can include boolean expressions, sequences and other sub-ordinate properties.

3.2.1 Sequences

One of the main requirements of an assertion language is to be able to concisely express design behavior over multiple clocks. PSL supports SEREs – Sequential Extended Regular Expressions to meet this requirement. It provides an easy and familiar way to engineers to capture sequential behavior. The syntax is derived from standard UNIX regular expressions and hence the name SERE. The first and foremost requirement of any temporal sequence is a neat way to move time forward. PSL uses "SERE concatenation" to achieve this. This operator is represented with a **semicolon** symbol " **;** ". Hence, the following pseudo-code:

```
{a;b}
```

Describes the following behavior:

- "a" being high in the current clock tick
- Wait till next clock tick (t+1) and
- Check for "b" being high.

The curly braces around the sequence mark the beginning and ending of a SERE. In real life, the delay between two such expressions can be:

- More than one
- A range
- Not necessarily occurring in contiguous clock cycles

PSL supports all these requirements via its repetition operators. There are three types of SERE repetition operators which are explained below.

3.2.1.1 Consecutive repetition operator

To be able to specify that a signal must be asserted continuously for say "3" clocks, one can write:

```
{sig_a;sig_a;sig_a}
```

PSL provides a short cut notion to the above as:

```
{sig_a[*3]}
```

The number can also be a range with a MIN & MAX specification as shown below:

```
{sig_a[*MIN:MAX]}
```

Few notes on the ranges:

- Both MIN & MAX have to be elaboration time constants.
- Both have to be natural numbers (0 and above).
- MIN can be set to 0.
- MAX can be set to the key word ***inf*** to indicate infinity.
- To specify one or more repetitions, a UNIX regexp style "+" can be used.

The following SEREs show few possible repetitions operators. They all capture the following requirement of a write protocol.

- The desired sequence starts when signal "wr_started" gets asserted.
- In the very next clock, wr_channel_busy gets asserted (The number of clocks that this signal remains asserted is variable, and few variances are shown below).
- The sequence finishes when a wr_done is seen after desired number of wr_channel_busy.

```
wr_started; wr_channel_busy[*2]; wr_done;
wr_started; wr_channel_busy[*0:100]; wr_done;
wr_started; wr_channel_busy[*2:inf]; wr_done;
wr_started; wr_channel_busy[+]; wr_done;
```

3.2.1.2 Non-Consecutive repetition operator

A non-consecutive repetition is very similar to consecutive repetition except that the occurrences of the repeated expression/sequence need not be contiguous. PSL uses the symbol "=" to denote non-consecutive repetition. The examples in the previous section with a non-consecutive repetition are shown below:

```
wr_started; wr_channel_busy[=2]; wr_done;
wr_started; wr_channel_busy[=0:100]; wr_done;
wr_started; wr_channel_busy[=2:inf]; wr_done;
```

3.2.1.3 GOTO Repetition operator

Some times, the requirement is such that, we want to "go to the nth repetition" and **immediately (1 clock after)** after the occurrence of that last repetition we would like to check for the next expression in sequence. The intermediate repetitions may be non-consecutive – i.e. can be spread across multiple cycles. This is referred to as GOTO repetition in PSL and is represented with "**->**" symbol. The examples in the previous section with a GOTO repetition are shown below:

```
wr_started; wr_channel_busy[->2]; wr_done;
wr_started; wr_channel_busy[->0:100]; wr_done;
wr_started; wr_channel_busy[-> 2:inf]; wr_done;
```

3.2.1.4 SERE within operator

The *SERE within* operator constructs a SERE in which one sequence's start and end points are fully "contained" within the other sequence. In the following example

```
A_Sere within B_Sere
```

A_Sere's start point should be after (or same as) the B_Sere and its end point should be before (or same as) that of B_Sere.

3.2.1.5 Compound SERE operators

While the repetition operators enhance PSL with the ability to build basic SEREs, more complex sequences can be described by combining two or more sequences. PSL provides the following operators for building compound SEREs.

- Fusion operator (:)
- SERE non-length matching and (&)
- SERE length-matching and (&&)
- SERE or operator (|)

Since these are advanced topics, they are not fully explained in this tutorial.

3.2.1.6 Endpoints

In PSL sequences typically span across multiple clocks. Sometimes it is useful to detect whether a particular sequence has reached its "end" regardless of its starting time. PSL supports it via endpoint construct. In PSL, endpoints are assigned a name by the user and such endpoints can be instantiated anywhere wherever a boolean expression is allowed and it results in true or false.

As an example, a packet transaction is considered complete only when every start-of-packet receives its end-of-packet signal. However, a packet can be aborted after one or

many transfers. The following endpoint captures the abort sequence via en endpoint and uses it in a property specification later.

```
endpoint e_pkt_aborted = {pkt_sop; pkt_xfer_in_progress[*1:100];pkt_abort};

property p_expect_eop = {pkt_sop |-> eventually! pkt_eop abort e_pkt_aborted};
```

3.2.2 Properties

In PSL, a Property forms the top level place holder for the expected design behavior. A property can instantiate named sequences, boolean expressions and other sub-ordinate properties and combine them using various property operators described in the following sections.

3.2.2.1 Suffix Implication operators

Many a class of design properties exhibit a "cause-and-effect" phenomenon, i.e. a property/sequence (condition-to-be-checked, or effect) is expected to hold only after a "triggering-condition" (or cause) occurs. This is also referred to as an "implication" operator. PSL provides two variants of implication operators that are described in the following sub-sections. A procedural way of understanding these implication operators is to visualize them as:

```
if (triggering-condition)
  condition-to-be-checked
```

Now, in a typical if construct, an else part is provided and is an interesting branch to verify as well, but in PSL, the else part is treated as "Vacuous success" – meaning that if the triggering-condition doesn't hold, don't bother to check any thing and return a TRUE. But this is not a real success and is treated as **vacuous success**.

3.2.2.1.1 Overlapping suffix implication (|->) operators

The condition-to-be-checked is checked in the SAME clock as the triggering-condition occurs. For example, to describe a property as: "Whenever there is a new packet (indicated by packet_start), the pkt_xfer_en_n should be low".

```
pkt_start |-> !pkt_xfer_en_n
```

3.2.2.1.2 Non-overlapping suffix implication (|=>) operator

This is a variant of the previous overlapping operator, and with this, the condition-to-be-checked is checked **ONE CLOCK after** the triggering-condition occurs. A simple example would be to check for a FIFO full:

```
fifo_almost_full |=> fifo_push |-> fifo_full;
```

The above property described that once the fifo is almost full, a push in the next clock cycle shall make the fifo_full to be asserted (in the same clock as the fifo_push occurred).

3.2.2.2 Occurrence operators

As mentioned in the introduction section, PSL provides few operators to specify "when to check for the property", in LRM they are referred to as "simple FL properties" and in this tutorial they are termed as "occurrence operators" – solely for the purpose of descriptiveness. PSL supports "***always, never, eventually!, next***" operators.

- ***always*** operator specifies that the property should hold in every cycle.
- ***never*** operator specifies that the following property should not hold during any cycle.
- ***eventually!*** operator requires that the property shall hold in some cycle in the future but before the end of verification process.
- ***next*** is a family of operators, basically expressing a requirement that the property following this operator shall hold in the next cycle(s).

3.2.2.3 until operator

An ***until*** operator requires that first property hold till the second property holds. e.g.

```
pkt_sop |-> !pkt_xfer_en_n until pkt_eop
```

The above property describes that once, a packet is started, the xfer_en_n shall be low until the packet ends.

3.2.2.4 before operator

A ***before*** operator requires that the one property hold before another. In a typical processor environment, an opcode_fetch process should occur before an execute_opcode phase, the following PSL property captures that requirement.

```
opcode_fetch before execute_opcode;
```

3.2.2.5 abort operator

In almost any practical property, there is a need to abort the checking in case of a reset, soft reset, pkt_abort etc. ***abort*** operator is intended to capture that intent. Using one of the previous packet transaction examples,

```
({pkt_sop} |-> {!pkt_xfer_en_n until pkt_eop}) abort pkt_abort;
```

The above property aborts checking once a pkt_abort is seen.

3.3 Verification Layer

3.3.1 Verification Directives

Properties are just declarations and they must be "directed" by a verification directive in-order to direct a verification tool to check/assume for the validity of the property. PSL supports the following directives:

- assert
- assume
- assume_guarantee
- cover
- restrict
- restrict_guarantee
- fairness
- strong_fairness

Among these, the *assert* and *cover* are the most frequently used in simulation based ABV. Other directives are mostly used in Formal Verification.

The *assert* directive instructs the tool to check that the property being asserted holds and if not to report a failure. The *cover* directive is used to specify valid legal behaviors and checking for their occurrence in simulation, in other words it is used to capture control centric functional coverage points.
PSL allows labeling of such directives and is a good coding style to use descriptive labels for them.

3.3.2 Verification Units

PSL is a property language that talks "about" the design and needs to be linked to a design unit (say RTL model) in order for a tool to check that the design meets the requirements as described by PSL properties. PSL supports a set of "verification units" as containers of properties so that a set of properties can be linked to a design. Of these, ***vunit*** is the most commonly used and is described below.

3.3.2.1 Using vunit

A ***vunit*** is used as a container for PSL properties.

```
 vunit v1 {

   property pkt_xfer = {pkt_sop; pkt_xfer_in_progress[*1:100];pkt_abort} @posedge
clk;

   a1 : assert pkt_xfer;
} // vunit v1
```

Now, in the above property, there are various signals used in the basic boolean expressions that are expected to be existing in a RTL design. The link to the design under test occurs via an argument to the ***vunit*** specification. These properties can be either bound to a design module or an instance of a module.

```
vunit v1 (module_name) {
//
}

vunit v2 (top.chip.ctl_unit.pkt_processor) {
//
} // v2
```

3.3.2.2 Adding properties "inline" in RTL.

Often RTL designers find it convenient to inline simple assertions, assumptions along with their RTL code. PSL LRM does not specify how this can be done, but a pseudo-standard is adopted by various EDA vendors to use a pragma based approach. In a Verilog RTL, one can use:

```
module proc_eng();
  reg clk,a,b;
  // psl a1 : assert always (a |-> b) @posedge clk;
endmodule
```

Similarly in a VHDL design, one can use "-- psl" pragma.

3.4 Modeling Layer

Though PSL is very capable of capturing certain classes of properties, it requires some additional helper code at times to model auxiliary combinational signals, state machines etc. that are not part of the actual design but are rather required to express the property in a concise manner. PSL uses underlying HDL to model such code. PSL LRM requires that such code be restricted to synthesizable sub-set of the HDL. A simple example is shown below in Verilog flavor.

```
vunit v1 (dut) {
// Signal a_and_b is purely for a verification purpose.
wire a_and_b = sig_a && sig_b;

a_and_b_ohot : assert always onehot(a_and_b) @ (posedge clk);
} // v1
```

4 Miscellaneous

4.1 Built in functions

PSL provides built-in functions to detect value changes, some of the commonly used functionalities etc. Most of these functions can be used inside property expressions.

4.1.1 rose(arg)

This function returns a Boolean result that is true when there is transition from 0 to 1. Else the result is false. Example:

```
default clock = (posedge clk);
property start_cell =
        always {rose (SOC)} |=> { payload; EOC};
```

4.1.2 fell(arg)

This function is opposite to that of rose. The result is true when there is transition from 1 to 0. Else the result is false.

4.1.3 prev(arg [,N])

This function returns the value of the argument in the previous clock. The argument can be any expression. The optional argument N specifies how many number of clocks to look back in the history and is defaulted to 1.

4.1.4 stable(arg)

This function returns a Boolean result. If there is no change in the value of the expression from the previous cycle then the result is true. Otherwise it is false.

4.1.5 countones(arg)

Counts the number of ones in the argument.

4.1.6 onehot(arg), onehot0(arg)

Returns true if the argument has exactly/atmost one bit set to 1.

4.2 comments

Comments can be added to PSL code using the same style as the underlying HDL. For example, in Verilog flavor one can use “//” for single line comments and “/*..*/” for multi-line comments. In VHDL flavor, “--“ is used to add comments.