Open Source Formal Verification

PSL simple constructs

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

July 2025

PSL

- PSL: Property Specification Language
- Language for specifying properties
- PSL flavors:
  - VHDL
  - Verilog
  - GDL
  - RuleBase (model checker)
  - SystemVerilog
  - SystemC

PSL layers

- PSL is made of 4 layers
  1. Boolean layer
  2. Temporal layer
  3. Verification layer
  4. Modeling layer

Boolean layer

- Boolean expressions written in the *flavour* language

VHDL examples

Let's have:

```
signal a: std_logic;
signal b: std_logic_vector(3 downto 0);
signal c: std_logic_vector(3 downto 0);
```

Then, the following are boolean expressions:

```
a
not a
b = "0001"
b = c
```

Temporal layer

- Temporal properties
- Represents a relationship between boolean expressions through time

VHDL examples

Let's have:

```
signal req: std_logic;
signal ack: std_logic;
```

Then, the following are temporal properties:

```
1. always (req -> next ack)
2. always {a;b} |=> {c;d}
```

1. Every request is immediately followed by an acknowledge
2. Every sequence *a*,*b* must be followed by sequence *c*,*d*

Verification layer (1)

- Directives indicating to the tools how to handle temporal properties

Examples

Let's have:

```
signal req: std_logic;
signal ack: std_logic;
```

Then:

```
assert always (req -> next ack);
```

Indicates that the tool (formal or simulation) has to check that each time a request is active, then the next clock cycle the acknowledgment signal must be asserted

Verification layer (2)

- This layer allows to group the PSL instructions into *verification units*

Examples

Let's have:

```
signal req: std_logic;
signal ack: std_logic;
```

Then:

```
vunit example {
    assert always (req -> next ack);
}
```

Is a verification unit (more to come later on)

Modeling layer

- Enable the use of the flavour language to perfom some computation

Example

```
vunit example_modeling {
    signal reading: std_logic;

    reading <= read_enable and chip_select;

    assert always (reading -> address/="ZZZZ");
}
```

Implication

- Most of the properties are based on implications
- For now, we just have the basic one:
  `->`
- `a -> b` simply means that `a` implies `b`

Temporal properties

- Temporal operators allow to create temporal assertions
  - `always`
  - `never`
  - `next`
  - ...
- A clock needs to be defined (pure combinational systems are not allowed in formal tools, but we'll see how to handle that)
  - Global, or
  - Specific to a property

Clock

- A clock needs to be defined

Global (shared by all properties)

```
default clock is rising_edge(clk); ← clk is a signal or a port
property prop_impl is always (a -> b);
```

VHDL

Specific to a property

```
property prop_impl is always (a -> b) @(rising_edge(clk));
```

VHDL

`assert` directive

- Once a property is defined, `assert` enables to check it holds

Example

```
property prop_impl is always (a -> b) @(rising_edge(clk));

assert prop_impl;
```

- `assert` can also directly include the property

Example

```
assert always (a -> b) @(rising_edge(clk));
```

Warning about reset handling: abort

- A reset is often the cause of a failing assertion
- As it changes the state of the system it is likely to jeopardize standard sequences
- `abort` enables to cancel the evaluation of an assertion on a condition

Example

```
property prop_impl is always (a -> b) abort rst;      <- rst aborts the always
property prop_impl is always ((a -> b) abort rst);    <- the evaluation restarts after an abort
```

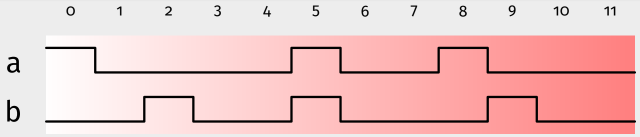
- This `abort rst` shall be present at the end of almost all properties
- For the sake of clarity the slides won't necessarily embed this statement

`always` and `never` operators (1)

- These two operators offer a way to indicate when a property shall be checked
- `assert always prop;`
    - The property must always hold
- `assert never prop;`
    - The property must never be observed
- `assert prop;`
    - The property must be true only at the first cycle of a trace
        - If not a pure VHDL `assert`

always and never operators (2)

Example



PSL

```
assert not(a and b);
```
← Pure VHDL assert

```
assert (a -> not b);
assert (b -> not a);

assert always not(a and b);

assert never (a and b);
```

`assume` directive

- An `assume` directive indicates that some property is assumed to be correct
- Typically used on the design inputs
    - Some combinations should never occur, as written in the specs
    - We consider a protocol from one point of view
- It helps the solver to reduce the search space
- Usually not only a help, but a requirement
- Example:
    - `assume never (a = b);` ensures that `a` and `b` will never be active at the same time
    - `assume waitrequest |-> stable(address)` ensures that the `address` is stable when required (Avalon bus)

`cover` directive

- It can be useful to know if a certain sequence has been observed (more to come later on)
- Hence the `cover` directive
- Example:
  - `cover {a;b;a} report "Seen";` in simulation would display a message when sequence *a*,*b*,*a* is observed

Good practice

- While not mandatory, `assert`, `assume` and `cover` directives can be labelled
- Easier to identify the failing assertion (for instance)

Example

```
property prop_impl is always (a -> b) @(rising_edge(clk));

assert_prop_impl: assert prop_impl;
```

Example

```
assert_direct: assert always (a -> b) @(rising_edge(clk));
```

PSL

LTL or SERE

- LTL-style : close to LTL (until, next, eventually, ...)
  - Linear Temporal Logic
- CTL-style : close to CTL (with existential operators), called Optional Branching Extension (OBE), but unavailable in formal tools
  - Computational Tree Logic
- SERE-style : exploits regular expressions
  - Sequential Extended Regular Expression
- Most of the LTL-style properties can be expressed thanks to SEREs

Useful functions

Function	Description
`onehot(exp)`	Returns true if one and only one bit of the expression is ’1’
`onehot0(exp)`	Returns true if at most one bit of the expression is ’1’
`isunknown(exp)`	Returns true if at least one bit is ’X’ or ’Z’. Not yet supported
`countones(exp)`	Returns the number of ’1’s in the expression. Not yet supported

Useful functions for time

Function	Description
`rose(exp)`	Returns true if called on a rising edge of `exp`
`fell(exp)`	Returns true if called on a falling edge of `exp`
`stable(exp)`	Returns true if the signal stayed stable for a clock cycle
`prev(exp)`	Returns the previous value of `exp`
`prev(exp,n)`	Returns the value of `exp` *n* cycles ago

PSL code placement

- PSL code can be placed in:
  - The VHDL code, as prefixed comments
  - The VHDL code, as is (with VHDL-2008)
    - Close to the code
    - Good idea?
  - A separate file, as a `vunit`
    - Definitely better for in/out ports
    - Better parallel working

PSL code placement : as comments

- Each statement shall start with `-- psl` (with a space in the middle)
- These statements shall be in the concurrent domain (not in a process)
- Every signal can be accessed
- In case of multi-line statement, only the first line shall be prefixed with `-- psl`

```
-- psl default clock is rising_edge(clk_s);
-- psl property myProp is always (a->b);
-- psl assert myProp report "Tragic mistake";
-- psl property myMultiLineProp is always
--      (a->b);
```

PSL code placement : In a vunit

- The `vunit` shall be linked to an entity or an architecture
- In case of an entity, only ports can be accessed
- In case of an architecture, signals can also be accessed

```
vunit my_unit(my_entity) {
  default clock is rising_edge(clk_s);
  property my_prop is always (a->b); ← a and b are ports
  assert my_prop report "Tragical error"; ← report only useful for simulation
}
```

```
vunit my_unit(my_entity(my_architecture)) {
  default clock is rising_edge(clk_s);
  property my_prop is always (a->b); ← a and b can be internal signals
  assert my_prop report "Tragical error";
}
```

Verification of a combinational component

- Formal verification requires a clock
- So... How about verifying a combinational component?
- Two options:
  1. Create a wrapper around the component, with an additional port for the clock
  2. Instantiate a clock signal in the PSL file
     - Works for proof and cover, not for bmc

Example

A *very* simple ALU

```
entity alu is
    port (
        a : in  std_logic_vector(7 downto 0);
        b : in  std_logic_vector(7 downto 0);
        m : in  std_logic;
        r : out std_logic_vector(7 downto 0)
    );
end entity alu;

architecture behave of alu is
begin

    r <= a or b when m = '0' else a and b;

end architecture behave;
```

Example

```
vunit psl_alu(alu) {

    signal clk : std_logic;

    -- All is sensitive to rising edge of clk
    default clock is rising_edge(clk);

    -- Parenthesis are important, else the "or" and "and" cause issue with
    -- the order of operations
    assert_mode0: assert always ((m = '0') -> (r = (a or b)));
    assert_mode1: assert always ((m = '1') -> (r = (a and b)));

    -- Identical, taking advantage of m automatic boolean translation
    assert_mode0b: assert always ((not m) -> (r = (a or b)));
    assert_mode1b: assert always (m       -> (r = (a and b)));
}
```

Combinational design: warning

- When checking a combinational design, be careful of the implication
    - $->$ is fine
    - $|->$ is also fine
    - $|=>$ will not detect errors if using the previous slide's solution