Open Source Formal Verification

Helper code and free variables

Yann Thoma

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

July 2025

Plan

1. Introduction
2. Helper code
3. FIFO example
4. FIFO - anyconst
5. FIFO - inputs in wrapper
6. FIFO - generate
7. Conclusion

Helper code

- Sometimes properties and sequences require more than just simple boolean expressions based on the available signals

⇒ the modeling layer enables more complex code to be part of it

  - In the `.psl` file
  - In the VHDL file if the PSL code is part of it

Helper code: when?

- Counting some events
  - Number of writes
  - Number of computations
  - Number of FIFO accesses
  - ...
- Use a signal assignment to do some computations that are used later on
- Doing some computation that do no fit into a SERE

Example

Example

```
vunit ... {

    signal pre_compute: unsigned(DATASIZE-1 downto 0);

    pre_compute <= (unsigned(a_i) + unsigned(b_i));


    assert always ((mode_i = "01") -> (pre_compute > 10));

    assert always ((mode_i = "10") -> (pre_compute < 25));
}
```

Example

Example

```
vunit ... {

    signal nb_write: integer := 0;

    process(clk_i) is
    begin
        if rising_edge(clk_i) then
            if write = ’1’ then
                nb_write <= nb_write + 1;
            end if;
        end if;
    end process;

    assert always ((nb_write > 3) -> (full_o = ’1’);
}
```

Helper Code - Restrictions

- In the helper code:
  - You can mix declarations, functions, procedures, processes, concurrent assignments
  - Stay to the synthesizable subpart of VHDL
  - Use it whenever it simplifies your code (and life)

Free variables

At some stage we may need to have a variable/signal free to take different values

Free variables are meant to help the verification process

Can be constrained

- Three options:
  - `anyconst`, `anyseq` (Specific to SBY)
  - inputs in a wrapper
  - `generate`

anyconst and anyseq

- The anyconst attribute tells the tool it can chose any value for the signal
- The anyseq attribute tells the tool it can generate any sequence for the signal
- In both cases, assume directives can constrain the values

Example

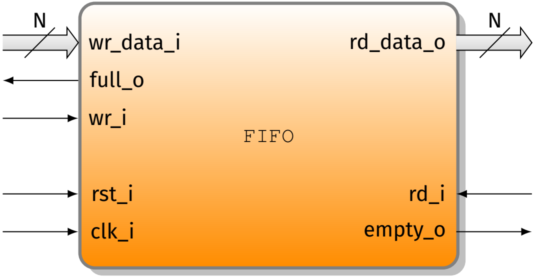
```
attribute anyconst : boolean;
signal s_data : unsigned(DATASIZE-1 downto 0);
attribute anyconst of s_data : signal is true;
signal s_seq : std_logic;
attribute anyseq of s_seq : signal is true;

assume always (s_data < 10);     ← Can not be higher than 9
assume never {s_seq;s_seq};      ← Can never be high two cycles in a raw
```

Verification of a FIFO

DUV

- Simple FIFO
- Synchronous read/write
- Two outputs for the FIFO status (empty/full)



Verification of a FIFO

Formal proof

- Properties to verify
  - P1. If the number of writes equals the number of reads, then `empty_o` shall be active
  - P2. If the difference between the number of writes and the number of reads equals the FIFO size, then `full_o` shall be active
  - P3. The nth data written shall be the nth to go out

A SystemVerilog digression

SystemVerilog assertion with local variable

```
property p_data_integrity;
    int cnt;
    logic[DATASIZE-1:0] data;
    @(posedge clk_i)
        (wr_i, cnt=wcnt, data=data_i) |=>
            (([0:$] (rd_i & (rcnt==cnt))) |->
             (data_o==data));
endproperty
```

- If there is a write, and it is the i^{th} write, then from now on, the i^{th} read from the beginning shall output the same data
- Quite elegant, but... no such possibility with PSL

Helper code

Two signals for counting the number or read and write

```
signal wcnt: integer := 0;
signal rcnt: integer := 0;

process(clk_i, rst_i) is
begin
    if rst_i = ’1’ then
        rcnt <= 0;
        wcnt <= 0;
    elsif rising_edge(clk_i) then
        if rd_i = ’1’ and empty_o = ’0’ then
            rcnt <= rcnt + 1;
        end if;
        if wr_i = ’1’ and full_o = ’0’ then
            wcnt <= wcnt + 1;
        end if;
    end if;
end process;

constant NB_ACCESS : integer := 4 * FIFOSIZE;
```

Helper code

Contraints on the number of read/write

```
constant NB_ACCESS : integer := 4 * FIFOSIZE;

assume always (wcnt < NB_ACCESS + 1);
assume always (rcnt < NB_ACCESS + 1);
```

FIFO

Checking full and empty

```
signal full_s: std_logic;
signal empty_s: std_logic;

full_s <= ’1’ when (wcnt = rcnt + FIFOSIZE) else ’0’;
empty_s <= ’1’ when (wcnt = rcnt) else ’0’;

assert_full: assert always (full_o = full_s);

assert_empty: assert always (empty_o = empty_s);
```

Example: FIFO - Parenthesis about over-constraining

- Let’s add the following assumption:

```
assume always (wr_i = ’0’);
```

- What happens?
  - All assertions will hold
  - Cover will fail

  ⇒ Usefulness of coverage

FIFO

Checking that the 4th data written is the 4th read

```
constant s_cnt : integer := 4;
constant s_data : std_logic_vector(0 downto 0) := "0";

p_data: assert always (
    ((wr_i =’1’) and (s_data = data_i) and (s_cnt=wcnt) and (full_o = ’0’)) |=>
    ({rd_i = ’0’;(rd_i = ’1’ and (empty_o = ’0’) and (s_cnt=rcnt))} |-> (data_o = s_data))
) abort rst_i;

p_data2: assert always (
    {((wr_i =’1’) and (full_o = ’0’)  and (s_data = data_i) and (s_cnt=wcnt));
     [*0 to inf];
     (rd_i = ’1’ and (s_cnt=rcnt) and (empty_o = ’0’))} |-> (data_o = s_data)
) abort rst_i;

p_data3: assert always (
    ((wr_i =’1’) and (s_data = data_i) and (s_cnt=wcnt) and (full_o = ’0’)) |=>
    next_event((rd_i = ’1’) and (empty_o = ’0’) and (s_cnt=rcnt))(data_o = s_data)
) abort rst_i;
```

FIFO

How to handle different values of `s_cnt` and `s_data`?

- Three options:
  - `anyconst`
  - inputs in a wrapper
  - `generate`

Example: FIFO - anyconst

Adding two free variables - replacing the constants

```
attribute anyconst : boolean;
signal s_data : std_logic_vector(DATASIZE-1 downto 0);
attribute anyconst of s_data : signal is true;
signal s_cnt : integer;
attribute anyconst of s_cnt : signal is true;

assume always (s_cnt < NB_ACCESS + 1);
```

Example: FIFO - inputs in wrapper

Wrapper

```vhdl
entity fifo_wrapper is
    generic (
        FIFOSIZE   : integer := 8;
        DATASIZE   : integer := 8);
    port (
        clk_i      : in    std_logic;
        rst_i      : in    std_logic;
        full_o     : out   std_logic;
        empty_o    : out   std_logic;
        wr_i       : in    std_logic;
        rd_i       : in    std_logic;
        data_i     : in    std_logic_vector(DATASIZE-1 downto 0);
        data_o     : out   std_logic_vector(DATASIZE-1 downto 0);

        -- For formal verification purpose
        s_data     : in    std_logic_vector(DATASIZE-1 downto 0);
        s_cnt      : in    integer);
end fifo_wrapper;
```

Wrapper

- In the wrapper: instanciate the `fifo` module
- In the `.psl` file: access `s_data` and `s_cnt` as previously

Example: FIFO - generate

Assertions in generate

```
gen_s_cnt: for s_cnt in 0 to NB_ACCESS-1 generate
begin
    ge_s_data: for value in 0 to 2**DATASIZE - 1 generate

        constant s_data: std_logic_vector(DATASIZE - 1 downto 0) :=
             std_logic_vector(to_unsigned(value, DATASIZE));

    begin
        -- Now we can use s_cnt and s_data
        p_data: assert always ...

        p_data2: assert always ...

        p_data3: assert always ...
    end generate;
end generate;
```

Free variables: Conclusion

- Wrapper is similar to the `anyconst` version
  - Use `assume` to restrict the values space
- Generate *creates* different versions of the assertions
  - Use custom generates to restrict the values space
  - Could be easier to identify where the error comes from
  - If the value is required in a `[*s_cnt]` time in a sequence

    ⇒ Would be good in a generate

    - However not supported by the tools