Processes with 'Incomplete' Sensitivity Lists and Their Synthesis Aspects.

Egbert Molenkamp, Gerhard E. Mekenkamp
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
the Netherlands
molenkam@cs.utwente.nl, mekenkam@cs.utwente.nl

Abstract.

Synthesis tools only support a subset of VHDL. In this paper we will focus on the synthesis aspects of processes with an incomplete sensitivity list. In general processes with a sensitivity list are used to describe combinational logic and clocked logic. The sensitivity list is called 'complete' when all signals which are read from within that process are in the sensitivity list, otherwise it has an 'incomplete' sensitivity list. Most, if not all, synthesis tools require that processes used to describe combinational logic should have a 'complete' sensitivity list, while for synchronous logic only the reset, if any, and clock signals should be in the sensitivity list.
Beside these two applications of processes with sensitivity list there is a vague support for other incomplete sensitivity lists, sometimes resulting in latches in the circuit and sometimes resulting in logic that has not the proper behaviour. This paper focuses on the synthesis aspects of processes with an incomplete sensitivity list, and will present a method how a subset of these processes can be synthesised, and also the problems synthesising processes with an incomplete sensitivity lists are discussed.

1. Introduction.

Take a look at the VHDL descriptions in figure 1. Does your synthesis tool support these four descriptions and if so, is correct logic generated?

We expect that your synthesis tool will have no problems with the description in the figures 1a and 1c. However the VHDL description of the figures 1b and 1d are probably not accepted or will result in erroneous logic. E.g. some synthesis tool generate an ordinary *and* gate for the description of figure 1b. During the synthesis process

```
process(clk, reset)
begin
 if reset='1'
  then q <= '0';
  elsif clk='1' and clk'event
   q <= d;
  end if;
end process;
```
fig. 1a

```
process(a)
begin
  q <= a and b;
end process;
```
fig. 1b

```
process(a,b)
begin
  y <= a or b;
end process;
```
fig. 1c

```
process(a)
begin
  y <= y xor b;
end process;
```
fig. 1d

Fig. 1: What does your synthesis tool support?

of the description of figure 1d a synthesis tool reported the warning "Port 'a' has no net attached to it" and generated erroneous logic, another tool reported "intermediate node 'Y' falls in a loop or is unreachable!" and generated no logic.

Most synthesis tools will, if logic is generated at all, generate erroneous logic for these kind of correct VHDL descriptions. Some people like to argue that VHDL descriptions similar to figure 1b and 1d are not correct.

0-8186-8180-2/97 $10.00 © 1997 IEEE

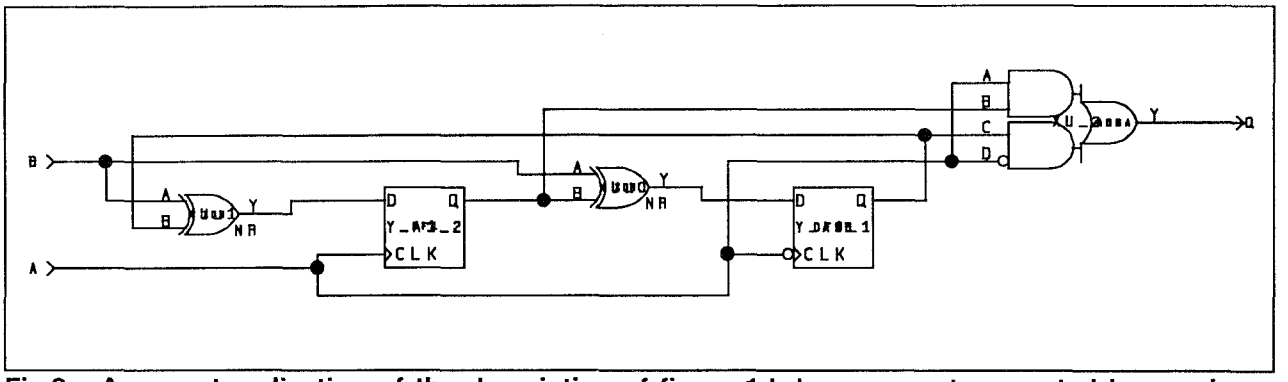

Fig 2: A correct realisation of the description of figure 1d; however not generated by any known synthesis tool!

However, from a specification point of view figure 1d nicely describes the behaviour of a circuit that performs 'an operation' on the rising and falling edge of the clock signal *a* (a correct realisation is shown in figure 2).

Via easy transformations on the VHDL description we can derive a synthesisable description which results in a correct description for a subset of the processes with an incomplete sensitivity list. In the next section this is illustrated using the figures 1b and 1d as examples. Next the problems are discussed if this method is generalised.

2. Two cases.

The rewriting rules are introduced using the cases that are not correctly synthesised by other synthesis tools as an example (figure 1b and 1d).

2.1 Case 1: design entity "funny".

Figure 3 gives a complete VHDL description of our first case. In this description the process triggers on signal *a*. Evidently this is not an ordinary *and* gate as synthesised by some synthesis tool. In fact only a change of signal *a* could potentially change the output *y*. The signal *a* has two possible values. Hence, it can be rewritten as shown in the architecture *first_step* shown in figure 4.

```
entity funny is
  port (a,b : in bit;
        q   : out bit);
end funny;

architecture behaviour of funny is
begin
  process(a)
  begin
    q <= a and b;
  end process;
end behaviour;
```

Fig. 3: design entity "funny" (figure 1b).

Rewriting rules:

1. For each possible value of the signal to which a process is sensitive a separate process is written that has a wait statement that detects an event to that value. In case the sensitivity list has more than one signal then for each possible combination of the signal values a process is added (see section 4.2.1).
2. The structure of the body is in essence not changed. But the signals assigned to are. For each signal which is assigned to an additional signals are introduced (in this example: *q_a1* in the case signal *a* is '1', and *q_a0* in case the signal *a* is '0'). Notice that sometimes variables in a process need a similar approach in case it uses the value of a previous execution. This will be discussed in the next section.
3. An additional concurrent conditional assignment statement is added for each signal that is assigned to. In that concurrent statements one of the additional added signals is selected (*q_a0* or *q_a1* in this example).
4. An insignificant change is made in the behaviour. The wait statement in the original description (figure 3) is implicitly at the end of the process, whereas in the translation the wait statements are placed at the beginning. This means that if the behaviour depends on the initialisation phase this rewriting rule is not valid. But seldom the hardware needed depends on the initialisation phase of the VHDL simulation cycle. The reason to move the wait statement to the beginning is that that is the style supported by synthesis tools.
5. Although the description is written in the style supported by synthesis tools, and indeed synthesised as shown in figure 4, it is not correct. Logic that contains data lines that changed the data and clock input of a flip-flop will probably result in an erroneous behaviour. Some synthesis tools report an error during the analyses phase.

Notice that for each possible value for which the process was sensitive a process is added. That means that in each separate process that signal (in general there are more signals to which a process is sensitive this is discussed in the next section) is a constant. Via constant propagation architecture first_step is reduced to the architecture shown in figure 5.

Rule 6:

Perform constant propagation using the known values of the signals in the wait statement.

The funny thing in the process labelled with *falling_edge* (figure 5) is that its output *q_a0* is always '0', but it is possibly synthesised with a register! (figure 5). Of course these constant values can be implemented more efficiently. The final description is shown in figure 6.

Rule 7:

Move signals in a process that are assigned to with a constant values outside that process.

2.2 Case 2: design entity "fun_xor".

In the first case a number of rules were given how to come from a process with an incomplete sensitivity list to a synthesisable description. In that case it was sufficient to add for each possible value of the signal a process that detects a rising edge of that value. Exactly one of these processes contain the valid data, and via a multiplexer (process labelled with *mux* in figure 6) that valid data is selected. But if there is a feedback, e.g. the signal assigned to in a process is also read in the same process, we cannot use this approach. This is illustrated with figure 7. Notice that signal y is assigned to and read from.

```
architecture first_step of funny is
 signal q_a1, q_a0 : bit;
begin
 rising_edge: process
 begin
  wait until a='1';
  q_a1 <= a and b;
 end process;

 falling_edge: process
 begin
  wait until a='0';
  q_a0 <= a and b;
 end process;

 mux:q <= q_a1 when a='1' else
          q_a0;
end first_step;
```

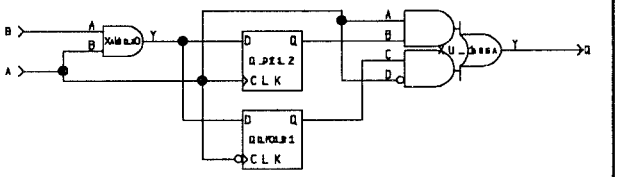

Fig 4: Result after the first rewriting, and its synthesis result.

```
architecture second_step of funny is
 signal q_a1, q_a0 : bit;
begin
 rising_edge: process
 begin
  wait until a='1';
  q_a1 <= b;
 end process;

 falling_edge: process
 begin
  wait until a='0';
  q_a0 <= '0';
 end process;

 mux:q <= q_a1 when a='1' else
          q_a0;
end second_step;
```

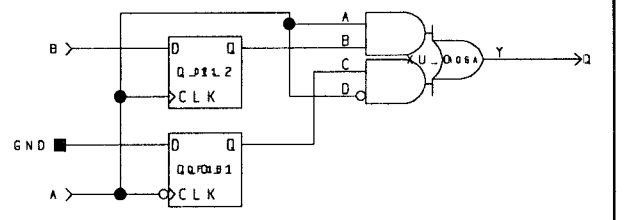

Fig 5: Result after the second rewriting, and its synthesis result.

```
architecture final_step of funny is
 signal q_a1, q_a0 : bit;
begin

 rising_edge: process
 begin
  wait until a='1';
  q_a1 <= b;
 end process;

 q_a0 <= '0';

 mux:q <= q_a1 when a='1' else
          q_a0;

end final_step;
```

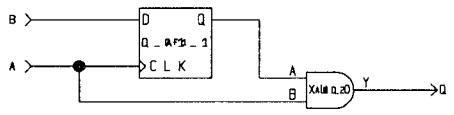

Fig 6: Result after the final rewriting, and its synthesis result.

```
entity fun_xor is
 port (a, b : in bit;
       q    : out bit);
end fun_xor;

architecture behaviour of fun_xor is
 signal y : bit;
begin
 process (a)
 begin
  y <= y xor b;
 end process;
 q <= y;
end behaviour;
```

Fig. 7: design entity "fun_xor" (figure 1d).

Figure 8 shows the result after applying the rules described in the first case. A similar problem occurs as is shown in figure 4; the data and clock lines depends on the same signal *a*, and synthesis tools will create erroneous logic or will not synthesise it. Notice that in the process in figure 7 signal *y* is read from and assigned to. In the rewriting rules a new signal was introduced for each signal assigned to (i.e. *y_a1* and *y_a0*). In fact an additional rule is required:

Rule 2a:

For each signal read from and assigned to in the same process, the signal read from should be replaced by the signal that corresponds with the previous value of that signal.

Let us explain this rule. Remember that exactly one process contains the valid data. E.g. in the second case, the process labelled with *falling_edge* has the valid data for signal *y* if signal *a* has value '0', and this valid data is represented with the signal *y_a0*. Figure 9 shows the result after applying this rule. However how is known that *y_a0* is the previous value, i.e. signal *a*'s previous value was '0'? In this case it is easy, since type bit has only two values, hence if signal *a* becomes '1', its previous value must have been '0'. In general this is a problem: the wait statement in the process labelled with *rising_edge* (figure 9) should be:

wait until a='1' and
"previous value of a is '0' ";

The latter is discussed in section 4.2.1

```
architecture not_correct of fun_xor is
 signal y : bit;
 signal y_a1, y_a01 : bit;
begin

 rising_edge:process
 begin
  wait until a='1';
  y_a1 <= y xor b;
 end process;

 falling_edge:process
 begin
  wait until a='0';
  y_a0 <= y xor b;
 end process;

 mux:y <= y_a1 when a='1' else
          y_a0;

 q <= y;

end not_correct;
```

Fig. 8: result after applying the rules of case 1.

```
architecture first_step of fun_xor is
 signal y : bit;
 signal y_a1, y_a0 : bit;
begin
 rising_edge:process
 begin
  wait until a='1';
  y_a1 <= y_a0 xor b;
 end process;

 falling_edge:process
 begin
  wait until a='0';
  y_a0 <= y_a1 xor b;
 end process;

 y <= y_a1 when a='1' else
      y_a0;

 q <= y;
end first_step;
```

Fig. 9: correct rewriting of design entity "fun_xor", and its synthesis result (figure 2).

3. Generalisation.

The ideas presented in the previous chapter are generalised in this chapter. For generalisation the following aspects are of interest:

- Variables used in the process.
- More than one signal in the sensitivity list.
- The type of the signal in the sensitivity list.
- Signals assigned to and read from are in the same process.

3.1 Variables used in a process.

Variables used in a process have two purposes. Variables that only contain intermediate values (i.e. a variable is assigned to before it is being read) or contain state information (the value of the variable of a previous execution of the process is used). Only variables that contain state information will influence the rewriting rules. The process with the sensitivity list is rewritten to a number of processes. As mentioned before there is exactly one process that contains the valid data, i.e. also the valid value for the variable. Suppose process p_i contains the valid data and the next process to execute is process p_j. How can the value of the variable in process p_i be assigned to the corresponding variable in process p_j. A solution is to make use of shared variables, since we know that only one of these processes is executed at a current time and delta. However, it is not expected that shared variables will be supported by synthesis tools. An additional signal is added for each variable that contain state information (figure 10). This additional signal is read from at the beginning of the process description, and assigned to at the end of the process description. In section 3.2 it is already explained how these 'signals' are taken care off.

```
rising_edge: process
  variable var : bit;
  ..
begin
 wait until a='1';
 var := signal_var_a0;
      -- signal_var_a0 contains value
      -- generated by the process that
      -- detected the falling_edge
      -- of signal a.

 .. the original description

 signal_var_a1 <= var;
      -- the signal signal_var_a1 is
      -- read in the process that
      -- detects the falling edge of
      -- signal a.
end process;
```

Fig. 10: How to handle variables that contain state information.

3.2 More signals in the sensitivity list.

Until now the sensitivity list of a process had exactly one signal of type bit. But what if a process has more signals in the sensitivity list? Again two cases can be distinguished (see figures 11 and 12). If a signal is read from and assigned to in the same process it has 'feedback'.

```
process (a,b)
begin
  q <= a xor c;
end process;
```

Fig. 11: process that without a 'feedback'.

```
process (a,b)
begin
   y <= y xor c;
end process;
```

Fig. 12: process with 'feedback'.

3.2.1 No feedback.

Figure 11 gives an example of a process that has no feedback. The rules used in section 3.1 can be applied here to. This will result in four processes that will have a wait statement similar as:

```
signal q_a0_b0, q_a0_b1,
       q_a1_b0, q_a1_b1 : bit;
begin
process
begin
 wait until a='0' and b='0';
 q_a0_b0 <= c;
end process;

process
begin
 wait until a='0' and b='1';
 q_a0_b1 <= c;
end process;

 ... etc.

q <= q_a0_b0 when a='0' and b='0' else
     q_a0_b1 when a='0' and b='1' else
     q_a1_b0 when a='1' and b='0' else
     q_a1_b1;
```

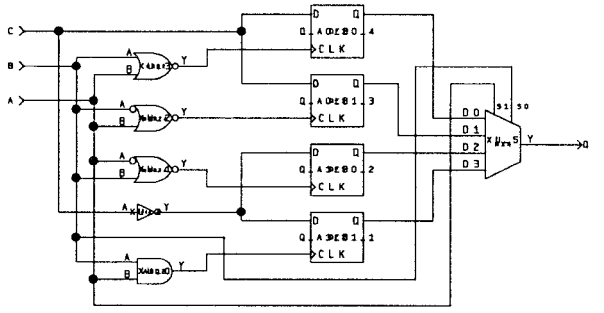

Fig. 13: Result after applying the rewriting rules of section 3.1.

wait until a='0' and b='1';

Constant propagation will result in the four processes from which two are given in figure 13. Also the synthesis result is shown. Most synthesis tools can not handle these 'complex' wait statements. Therefore an additional signal for each condition is added, like:

signal a0b1 : bit;
..
a0b1 <= (not a0) and b1;
..
wait until a0b1='1';

Hence multiple signals in the sensitivity list will in general result in *gated clocks*. In general a risky solution since the synthesis tool will not create hazard-free logic for these gated clocks.

We may conclude that multiple signals in the sensitivity list will in general result in gated clocks.

3.3 Type of the signal in the sensitivity list.

What if the type of the signal is of another type than type bit? In fact we have to distinguish the logic types (bit, std_ulogic, std_logic, boolean, and other types with only two possible values) and other types.

For other types it will become rather difficult to find easy rewriting rules. Suppose a process triggers on an integer value *int*.

process (int)
begin
..
end process;

According to the rewriting rules of section 3.2, and remember that the integer has the minimal required range according the language reference manual of VHDL the huge amount of 2**32 processes are needed! Whereas each wait statement is of the form:

wait until int = 45;

And this is still at an abstract level. E.g. an integer is represented at the logical level using a bit_vector of 32 bits wide. E.g. at the logic level this will have the form (vector *v* represents the integer *int*):

wait until .. v(2)='1' and v(1)='0' and
v(0)='1';

This again results in gated clocks (see section 4.2.1)! In special cases, similar to that of figure 11, this can be reduced.

If an enumerated type (other than a logical type) is used to trigger a process this type also has to be coded at bit level resulting in the same problem.

We may conclude that in general a signal, that has another type than a logical one, will result in gated clocks using the previous rewriting rules.

3.4 Signals assigned to and read from are in the same process.

Section 3.2 discussed this problem (figure 7). In this description signal *y* is read from and assigned to in the same process. An additional rewriting rule was required (rule 2a). If the sensitivity list only contains a signal of a logical type, or other types with only two value, we have already shown that the translation is easy (see figure 9).

However, assume that a process triggers on an integer value *int* which type is bound between the ranges 0 and 3 (see figure 14 as an example). Notice that due to the fact that there is a feedback, signal *y* is read from and assigned to, also the previous value of signal *y* is needed. Due to the rewriting rules this will result in the description given in figure 15.

```
process(int)
begin
 y <= y or b;
end process;
```

Fig. 14: process has a type other than a logical type in the sensitive list.

```
process
 begin
  wait until int=0 and prev_int=1;
  y_int0 <= y_int1 or b;
 end process;

 process
 begin
  wait until int=0 and prev_int=2;
  y_int0 <= y_int2 or b;
 end process;

 process
 begin
  wait until int=0 and prev_int=3;
  y_int0 <= y_int3 or b;
 end process;

 process
 begin
  wait until int=1 and prev_int=0;
  y_int1 <= y_int1 or b;
 end process;

.. etc.

 y <= y_int0  when int=0 else
      y_int1  when int=1 else
      y_int2  when int=2 else
      y_int3;
```

Fig. 15: part of the result after rewriting the description of figure 14.

How to determine the previous value of signal *int*? The predefined attribute last_value can not be used since it is not supported by synthesis tools. The following process could be used to remember the last value:

```
process (int)
begin
 prev_int <= tmp;
 tmp <= int;
end process;
```

In fact this is the kind of processes focusing on in this paper, and in section 4.3 it was already concluded that if the signal *int* is of another type than a logical one it will in general result in gated clocks in the realisation.

We conclude that the rewriting rules only will result in correct logic if the signal in the sensitivity list is of a logical type, or of a type with only two values.

4. Conclusion.

Synthesis tools only support a subset of VHDL. In this paper we focused on the synthesis aspects of processes with an incomplete sensitivity list. If a reset signal, if any, and a clock signal is in the sensitivity list a synthesis tool will create synchronous hardware if the description is of the form:

```
process(reset,clk)
begin
 if reset='1'
  then -- resetting
 elsif clk='1' and clk'event
..-- (or rising_edge(clk) )
..
```

This paper has shown that in case there is only one signal in the sensitivity list, and that signal is of a logical type (bit, std_logic..) or of a type with only two values, than there are easy rewriting rules available that result in correct synthesised logic.

If there are more signals in the sensitivity list, or the only signal available is of another type than the logical one, gated clocks are created. Since synthesis tool will commonly not create hazard-free logic for the logic in the clock line this will result in erroneous logic.