SIL: an Intermediate for Syntax Based VHDL Synthesis.

Egbert Molenkamp, Gerhard E. Mekenkamp, Jaap Hofstede, Thijs Krol
Department of Computer Science
University of Twente
PO Box 217
NL 7500 AE Enschede
the Netherlands
email: molenkam@cs.utwente.nl

Summary

It is well known that VHDL synthesis systems do not accept the complete IEEE Std 1076-1987/1993. The intention of this paper is to learn about the necessity of synthesis guidelines, or more precise their irrelevance. Most VHDL users are familiar with the problem that a given synthesisable VHDL description for synthesis system X is probably not accepted by synthesis system Y.

For reasoning about a VHDL description an intermediate format based on single token graphs is used. This intermediate language, that is developed in the ESPRIT SPRITE Project 2260, is called SIL: Sprite Input Language [KL92].

It is not the intention to show how the complete VHDL language is translated to SIL. A simple example is used to show that different VHDL descriptions with the same behaviour will indeed result in the same SIL graph, hence in the same hardware. The mapping of a SIL graph to hardware is outside the scope of this paper.

1. Introduction

Figure 1 shows the use of SIL as an intermediate representation between specification languages and silicon compilers. In case VHDL is used as specification language, VHDL constructs are translated to SIL; except the 'delay' related topics in VHDL. Currently most synthesis systems use a semantic approach when they accept a VHDL source file for synthesis. E.g. the statement "***wait until*** *clk='1'* " often means that signal *clk* is a clock input of a flipflip. Often only a subset of VHDL is accepted by a synthesis system: the synthesis guidelines. However it is possible to have a VHDL description with the previous statement that has no memory effect at all.

In our approach we use a syntax based translation: we do not interpret statements during translation. SIL is a convenient format to perform transformations on. Since we do not interpret the VHDL source file during the translation to SIL it is necessary that the delta delay mechanism is also translated to the SIL graph.

First a short introduction of SIL is given. Then it is globally shown how VHDL is translated to SIL. Finally we will illustrate that indeed different VHDL descriptions result in the same graph using a VHDL description of a 'wire' as an example.

Figure 1: SIL, an intermediate between specification languages and silicon compilers

2. An introduction to SIL

A complete description of SIL is found in [Kl92]. SIL is a graph that contains nodes and edges between nodes. SIL is based on the single token flow concept. During an implicit clock period each node fires exactly once. If a node has input access points then it only fires if on each input access point a token is received (an exception is the delay node that will be discussed later).

Two kinds of nodes can be recognised:

- primitive nodes
- hierarchical nodes

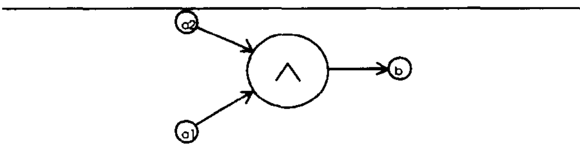


Figure 2: A logical *and* node (a primitive node)

The behaviour of primitive nodes is defined in the language. Hierarchical nodes contain primitive nodes or hierarchical nodes. Even recursion of hierarchical nodes is allowed.

Two kinds of edges are used in a SIL description:

- data flow edges
- sequence edges

Data flow edges contain the data (in a token) that goes from an output access point of a node to an input access point of a node. Sequence edges are only used to force an ordering of execution. If there is a sequence edge pointing from node *i* to node *j*, then node *j* fires after node *i* has fired.

The node in figure 2 is an example of a primitive node, the *and* node. After both tokens have been received at the input of the node an output token is produced that contains the logical *and* of both input tokens. A primitive node may be decorated with sequence edges and conditions (the latter is discussed later).

In figure 3 for a piece of sequential VHDL code the SIL graph is given. Due to the sequential order the first statement is executed before the second statement, i.e. there is a sequence edge (the dashed line) between the two *and* nodes.

In this description the second node may only fire if the tokens from the data flow inputs and sequence input are received. Notice that in this example the sequence edge can be removed (a transformation on the SIL graph) since the data flow edge *d* guarantees the correct ordering of execution.

A node can also have a condition access point. Figure 4 illustrates the use of such an access point. The value received at input access point *c* depends on the condition *cond*. In a SIL graph a node can be decorated with a condition access point. These condition access points are small circles attached to a node for a true-condition and a small bullet for a false-condition. A true-condition gives a *grant* upon the reception of the value true, and a *deny* upon the reception of

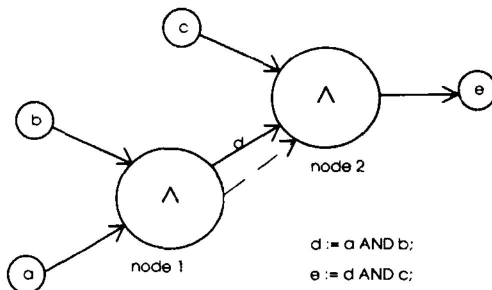


Figure 3: SIL graph for a piece of sequential VHDL code

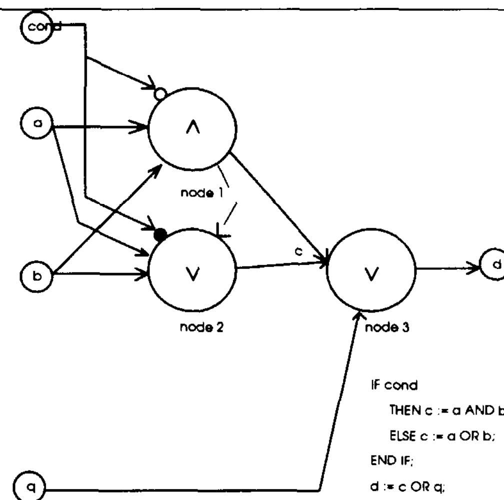


Figure 4: SIL graph with conditions

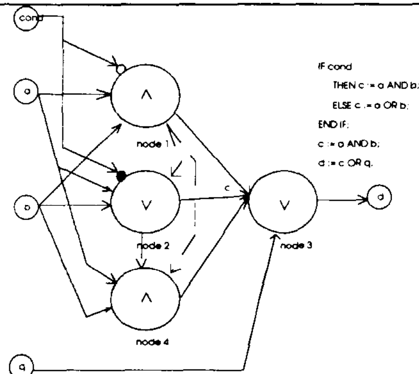


Figure 5: SIL graph with conditions and sequence edges

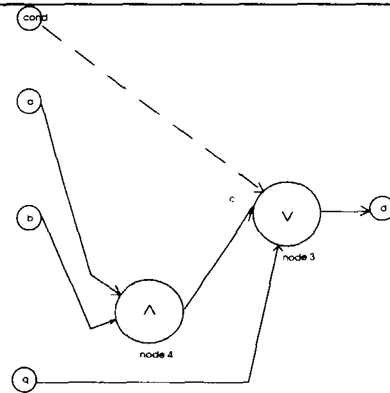


Figure 6: SIL graph after transformations

the value false. A false-condition has the opposite behaviour. Execution of a conditional node is granted if it receives a grant upon all its conditions, otherwise the execution is denied. If the condition input *cond* fires a true then the logical *and* of both inputs should be received at input access point *c*, otherwise the logical *or* of both inputs. However, it is required that each output access point exactly fires once, e.g. what token should node 2 fire if the condition is true? Hence, an empty token! A node with a condition access point is the only node that can fire an empty token and it fires an empty token if and only if the conditions are not fulfilled.
In figure 4 also merging of data flow edges is shown at input *c* of node 3. In case of a join of more than one data flow edge, the value that is used is the last non-empty value that is received at the access point. If all received tokens are empty the used value is an arbitrary element of the type. When the value of the edge is empty an arbitrary value of the type is taken.
For the previous example exactly one of the nodes node 1 and node 2 fires an non-empty value so the received value at the input access point of node 3 has always the same value as in the VHDL code.
Figure 5 shows a more complex example in which the sequence edges are necessary. Analysing the VHDL code it is easy to show that the condition part is superfluous. Let us now see how this is detected using transformations on the SIL graph. The VHDL code is translated into a SIL graph. The two sequence edges pointing to node 4 guarantee that this node always fires after firing of node 1 and node 2, therefore the token of node 4 is always the last received token at the input access point *c* of node 3. Since node 4 has no condition access point the token is always a non-empty one. Hence, the received value at input *c* of node 3 is always the value send by node 4. The behaviour of this SIL graph is equivalent with that of the graph in figure 6. From the previous example it is clear that the linear ordering of the fired tokens is essential in SIL. The sequence edge between *cond* and node 3 is necessary. The ordering between *cond* and *d* should be preserved after transformation, hence the sequence edge.
If a node fires an empty token and that token is send to more than one input access point then each input access point determines an arbitrary value for the received empty token. Hence, the values are not necessarily the same.

3. Translation of VHDL to SIL

Each VHDL process suspends execution only if it encounters a wait statement. The extreme situation is reached when a process has no wait statement at all. The translation of such an infinite execution of a process is described in the next paragraph. Thereafter it is shown how a wait statement is added to the translation procedure. Finally it is shown how multiple processes can be translated. For our simple example in chapter 4 it is not necessary to study the translation of multiple processes.

3.1. The translation of a VHDL process without a wait statement

Figure 7 shows a VHDL description without wait statements. This VHDL process will never stop executing. Still this is a very interesting description, since in general a process description can loop multiple times so this

```
entity endless is
end endless;

architecture test of endless is
 signal c : natural;
begin
 p:process
  variable b : natural;
 begin
  b := b + 1;
  b := b + 1;
  c <= c + 1;
  c <= c + 1;
 end process;
end test;
```

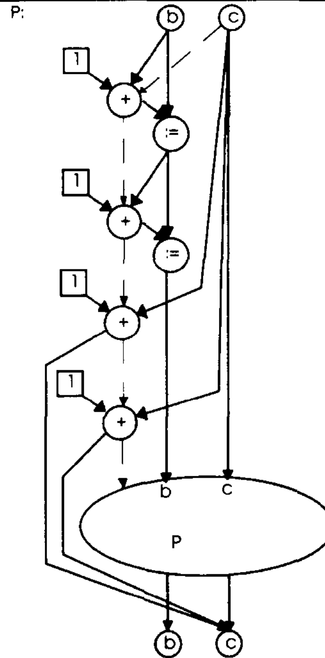


Figure 7: VHDL description without a wait statement and the corresponding SIL graph for the process

should be translated to SIL. In figure 7 the VHDL description *endless* is translated into a SIL graph using a recursive instantiation of node *p*. This process has two inputs variable *b* and signal *c*. In the corresponding SIL graph the execution ordering of the nodes is prescribed by the sequence edges (dashed lines). After the four additions are performed node *p* is executed again, and this will never stop! In the next chapter it is shown how a wait statement stops the recursive instantiation.
The difference between variables and signals is also apparent in this figure. The first adder in the graph adds 1 to the input value *b* and assigns it to the assign node (:=). (The assign uses the last non empty value, since the add node has no condition access point it is the value of b+1. See explanation of the join in the previous chapter. In the next chapter it is shown that the join makes sense (figure 8).) The second add node again adds 1. The input value of the node *p*, in graph *p*, is input value of *b* plus 2. Since the value of a signal does not change immediately each node uses the input value of *c*, even the node *p*. All new values assigned to signal *c* are joined in the output node *c*. Remember that this output node will accept the last non empty value.

3.2. The translation of a VHDL process with a wait statement

An infinite execution of a process in a VHDL description results in an infinite recursion of the SIL graph (figure 7). More interesting is the process with a wait statement. For each wait statement in a process a control line is added (*exe* in figure 8). These control lines are implemented the same way as a variable. The initial values of these control lines are true (this is discussed in the next chapter). This is equal to a VHDL description in which initially each process is executed. The process should execute the statements until it encounters a wait statement. In the SIL graph this is implemented by adding a true condition access points at the nodes for each control line. In the SIL graph the wait statement uses the previous value of a signal to detect an event (*a-* is the previous value of *a*).
If input access point *exe* has the value 'true' the nodes of the SIL graph including the node *wait* are executed. In this wait node the node *neq* is decorated with a bullet. Since *exe* has the value 'true' this node fires an empty token. The last non empty value received at the assign node is 'false'. Hence, the value of *exe* is 'false' and the statements after the *wait* node will not execute. This process has finished its execution at the current delta. Other processes, not shown in the figure, perform similar actions.

```
entity wait_statement is
  port (x, y : in bit;
      c, d : out bit);
end wait_statement;

architecture test of
      wait_statement is
  signal a : bit;
begin
  p:process
    variable b : natural;
  begin
    b := b + 1;
    c <= c + 1;
    wait on a;
    d <= d + 1;
  end process;

  q:process
  begin
    a <= x and y;
    wait on x,y;
  end process;
end test;
```

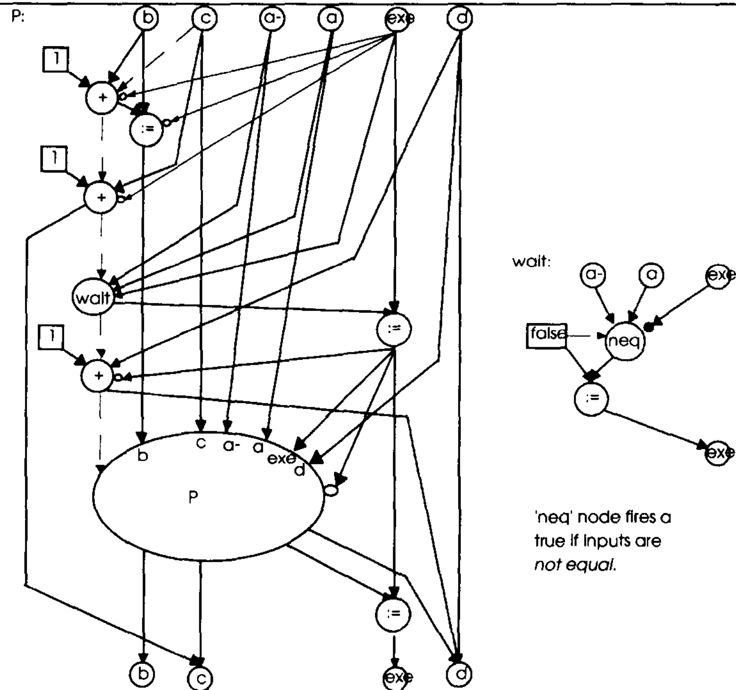


Figure 8: VHDL description with a wait statement and the corresponding SIL graph for process *p*.

After all processes have been suspended the signals are changed at the next delta delay. Now the SIL graphs of the processes are executed with the new signal values. The previously calculated values of the variables and control lines are used. Hence, *exe* has the value 'false'.

Now two cases are recognised:

- **Signal a is changed.** The SIL graph is executed. Since *exe* has the value 'false' the first nodes are not executed. This is correct since the process should start with the execution of the statement after the wait statement where it was suspended. In the *wait* node the node *neq* fires a 'true' since *a-* and *a* are different (signal is changed!) and the value at the condition access point is 'false'. The last received value at the assign node in the *wait* node is 'true'. Hence, the wait node fires a 'true'. The add node beneath this wait node is executed, and node *p* is executed. The first nodes in this node *p* are executed (this is similar in VHDL by running the statements at the top of the process description) until it encounters a wait again.
- **Signal a is not changed.** The SIL graph is executed. The first nodes are not executed due to the false condition of the *exe*. The wait node is executed, however since *a-* and *a* are equal this node fires a 'false'. Hence, the other nodes are not executed. In this case the VHDL process was not executed also.

From the previous it is clear that also the initialisation phase is correctly translated to SIL. It is left as an exercise:

- That the recursive use of node *p* is easy unfolded since only one extra instantiation of the node is necessary.
- To draw the SIL graph that corresponds with process *q* in figure 8. In this graph the wait node has to check for an event on *x* and/or *y*. In figure 9 the access points of this graph are given.

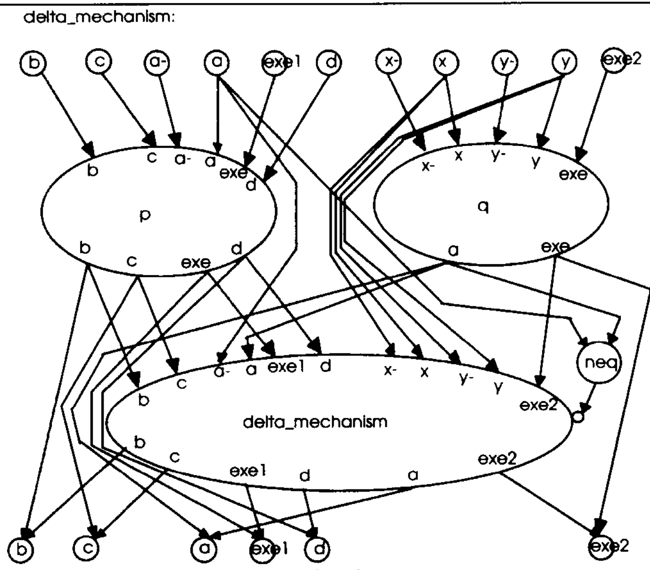


Figure 9: The delta delay mechanism in SIL for figure 8.

Until now only one process is translated. Within one delta cycle there is no interchange of information between processes, since the values of the signals are constant. In the next chapter the interaction between processes is discussed. At this level the delta mechanism is described.

3.3. Translating multiple VHDL processes to SIL

Like other synthesis tools explicit timing information is not supported for synthesis in our case. Statements like "y<=a after 10 ns;" are handled like "y<=x"; and a warning is generated. This means that a circuit is stable if no events are generated for the next delta. It is essential to check whether a process has changed a signal or not. If all signals are stable the SIL graph is not executed again. In the SIL graph after each delta the old and new signal values are compared. When a signal is changed the SIL graph is executed again; a recursive instantiation. Figure 9 Shows the SIL graph for the VHDL description with the delta delay mechanism of figure 8. In this figure each recursion of node *delta_mechanism* is equal to a delta step in VHDL. Hence, the newly generated signal values by the processes *p* en *q* are input of the next instantiation. During the delta steps the primary input signals are not changed (signals *x* and *y*). After the first delta these signals are stable and do not force a new delta step. In this example only signal *a* can force a new delta step. Thus if the value of the

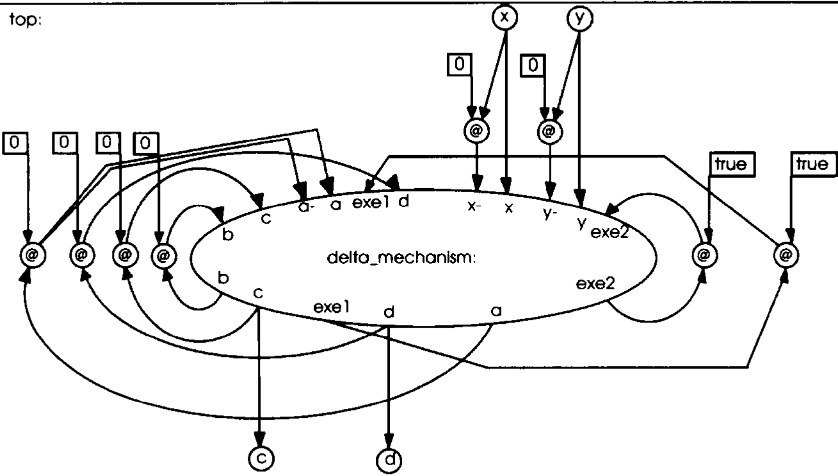


Figure 10: The top level description.

input access point *a* and the generated output value for *a* by process *q* are different a next delta step is necessary, that results in a recursive call of node *delta_mechanism* due to the true condition access point.

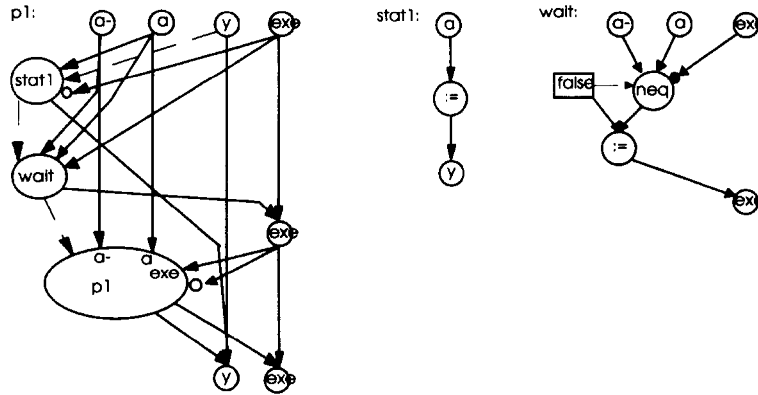
Now almost the top level description of the SIL graph is described. After the circuit is stable at the current time it should remember the values of the signals and the variables used in processes for the next execution. Furthermore the initial values of objects in VHDL should be taken care off. Both are implemented in SIL using the delay node. Figure 10 gives the top level description. Some remarks:

- **Primary input.** To detect an event on a signal the new and the previous values are required. The delay operator is used for this. The initial value is taken care off by the constant node at the input of the delay operator. Notice that the primary input *x* of the node *delta_mechanism* is connected to *x-* and *x* at the recursive node *delta_mechanism*, since the processes *p* and *q* do not assign to both signals.
- **Internal signals and variables in processes.** These objects have to remember there last value. For the initial values again constant nodes are used. Both are implemented with the delay nodes.
- **Control lines for the wait statement.** At the beginning all processes execute. Hence the initial values of the control lines (*exe1* and *exe2*) are 'true'. After the initialisation the control signals should have the previous values. This assures that a process starts at the wait statement where it was suspended.

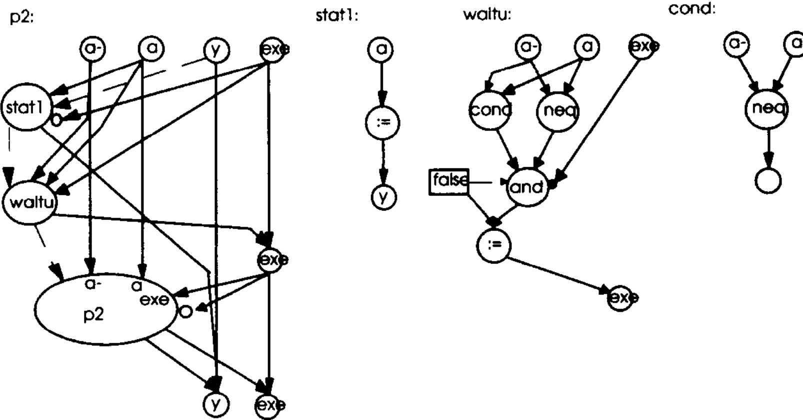
4. Example: Different VHDL descriptions of a wire

```
entity wire is
 port (a : in bit; y : out bit);
end wire;

architecture behaviour_1 of wire is
begin
 p1:process
    begin
     y <= a;
     wait on a;
    end process p1;
end behaviour_1;

architecture behaviour_2 of wire is
begin
 `p2: process
    begin
      y <= a;
      wait on a until a'event;
    end process p2;
end behaviour_2;

architecture behaviour_3 of wire is
begin
    p3:process
    begin
     y <= a;
     loop
       wait on a;
       exit when a'event;
     end loop;
    end process p3;
end behaviour_3;

configuration wire1 of wire is
 for behaviour_1
 end for;
end wire1;

configuration wire2 of wire is
 for behaviour_2
 end for;
end wire2;

configuration wire3 of wire is
 for behaviour_3
 end for;
end wire3;
```

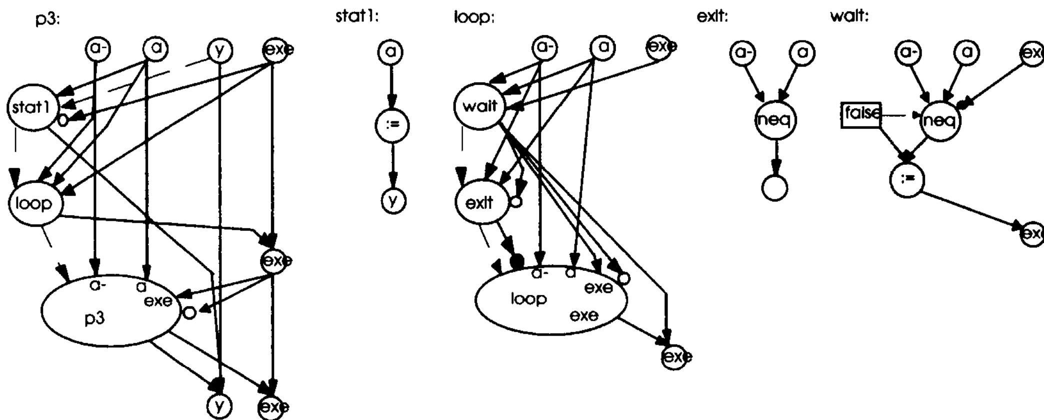
Figure 11: Three architectures with the same behaviour: a 'wire'.



a) SIL graph for process p1



b) SIL graph for process p2



c) SIL graph for process p3

Figure 12: The SIL graphs for the three different processes of figure 11.

Three different VHDL descriptions with the same behaviour are given in figure 11. Most synthesis tools can not handle all three VHDL descriptions. Since each architecture contains one process only the SIL graphs of the three processes are shown. Figure 12a gives the SIL graph for process *p1*. It will be shown that the SIL graphs of the other two process descriptions can be easily translated to the same SIL graph as figure 12a:

- **Process *p2* (figure 12b).** The node *waitu* detects an event on signal *a* and if there is an event the condition should be 'true' before it resumes execution. The SIL graph for the node *cond* is again a comparison between *a-* and *a*. Hence the SIL graph at

both inputs of the and node (in graph *waitu*) are equal. The graph *waitu* is reduced to the same graph as graph *wait* in figure 12a. With this simple transformation it is shown that the graphs for process *p1* and *p2* are equal.

- **Process *p3* (figure 12c).** In the VHDL description a loop statement is translated into a SIL graph using again a recursive instantiation of the loop, since in general a loop is executed multiple times. There are two condition access points at node *loop*. The true condition access point has the function described earlier. The second condition is caused by the exit statement. If the condition in this VHDL statement is 'true' the loop should not execute again, hence the false condition at node loop (small bullet). Node *loop* is only executed if there is a 'false' at the false condition access point and a 'true' is received at the true condition access point. There is only a 'false' at the false condition access point of the loop if *a-* and *a* are equal (graph *exit*). However, in that case the true condition access point always receives a 'false'. Hence, node *loop* in graph *loop* is never executed. The node *loop* can be removed, and the *node* exit also makes no sense anymore. Finally for the node *wait* in the graph *loop* the graph *wait* is instantiated again, and the result is the same as figure 12a.

5. Conclusion

The VHDL synthesis guidelines required by several synthesis tool are often superfluous. This paper shows that it is possible to translate VHDL code to an intermediate format, including the delta delay mechanism. On this intermediate format a number of rather simple transformations are performed. This way it is possible to accept a wide range of VHDL descriptions that. VHDL descriptions with the same behaviour result in the same hardware.

In this paper this is illustrated using three totally different, and indeed exotic, examples of a VHDL description that describe a wire.

6. Literature

Be93 Bergemaschi R.A., *High-Level Synthesis in a Production Environment: Methodology and Algorithms*, Fundamentals and Standards in Hardware Description Languages, Kluwer Academic Publ. 1993, pag. 195-230, edited by J. Mermet

En93 Engelen W., P.F.A. Middelhoek, C. Huijs, J. Hofstede, Th. Krol, *Applying Software Transformations to SIL*, SPRITE deliverable LS.a.5.2./UT/Y5/M6/1A, June 1993 (available at http://www.spa.cs.utwente.nl/aid/)

Hu94 Huijs C., Th. Krol, *A Formal Semantic Model to Fit SIL for transformational Design*, Proc. of the 20th. EuroMicro Conference, pp. 100-107, Liverpool, September 1994

IE88 IEEE Standard VHDL Language Reference Manual, Std 1076-1987, Published by the IEEE, New York 1988

Kl92 Kloosterhuis W.E.H. et. al., *The SPRITE Input Language SIL-1, Language Report*, SPRITE deliverable LS.a.a./Philips/Y3-M12/2, October 1992 (available at http://www.spa.cs.utwente.nl/aid/)

Kr92 Th. Krol, J. van Meerbergen, C. Niessen, W. Smits, J. Huisken, *The SPRITE Input Language: An Intermediate Format for High-Level Synthesis*, proc. of EDAC 92, pp. 186-192, Brussels, March 1992.

Me93 Mekenkamp G.E., *Translating VHDL with the delta delay mechanism to SIL*, internal report, University of Twente, the Netherlands.

Mi94 P.F.A. Middelhoek, *Transformational Design of Digital Signal Application*, Proceedings of the ProRISC/IEEE Workshop on CSSP, pp. 176-180, Arnhem, March 1994