A Syntax based VHDL to CDFG Translation Model for High-Level Synthesis

Gerhard E. Mekenkamp, Peter F.A. Middelhoek, Bert E. Molenkamp, Jaap Hofstede, Thijs Krol

University of Twente
Dept. of computer science
P.O. BOX 217, 7500 AE Enschede, The Netherlands
email: mekenkam@cs.utwente.nl

Abstract

In this paper a model is presented that allows translation of close to full VHDL (excluding time constructs) to a control-data flow graph. The syntax based translation leaves the VHDL execution mechanism in the graph. Behavior preserving transformations are used to remove the execution mechanism.

1. Introduction

Currently VHDL is often used as a specification language for high-level synthesis. However, VHDL was originally designed for simulation purposes. Its event mechanism makes high-level synthesis on VHDL difficult. In [Cam91] difficulties in synthesising sequential VHDL descriptions are identified, e.g. time specifications and wait-statements. Existing synthesis tools try to reveal the meaning of a VHDL description by searching combinations of statements that can be mapped to hardware. For instance a process with all signals in its sensitivity list is recognized as combinatory logic. This approach, which we refer to as a semantics based translation, can get quite complicated. For example when does a variable in combination with the statement \textit{WAIT UNTIL clock='1'} require a register? Furthermore the number of combinations that can be recognized is very large. Therefore, synthesis guidelines are given that restrict VHDL to a synthesisable subset. Since different synthesis systems restrict VHDL in different ways, designs accepted by one synthesis tool might not be accepted by another tool. An other problem of the semantics based translation is that even small syntactical variations can lead to different implementations. A small variation in the VHDL description can result in an implementation that is much smaller or larger. Designers check the results of synthesis and if necessary adapt the VHDL description. These steps are continued until the constraints are met. Each time the VHDL description is altered extensive simulation, which is very time consuming, should be used to check whether the change was correct.

In our system, called TRADES (Transformational DEsign System [Mid94a][Mid94b]), VHDL is translated to a control-data flow graph (CDFG) on a per statement basis which we refer to as a syntax based translation. Due to the syntax based approach the VHDL event mechanism appears in the CDFG (our CDFG is called SIL [Klo92]), but less synthesis guidelines are required. The syntax based translation is straight forward this enables the designer to recognize the design in the graph. Furthermore the implementation suggestion contained in the VHDL description remains intact. Using pre-proven behavior preserving transformations the VHDL event mechanism can be removed. Thereafter the designer can optimize the SIL graph using behavior preserving transformations (see figure 1). Finally the transformed graph can be mapped on hardware. This eliminates the need to alter the original VHDL description if synthesis results are not satisfying.

![Figure 1. Design flow](image)

In the following section we will first explain our transformation based design methodology. In section 3 we will describe the SIL design
representation. Then the translation model is described in section 4. Next a way to remove the VHDL event mechanism from the SIL graphs using behavior preserving transformations is given in section 5.

2. Design Methodology

In high-level synthesis the designer starts from a high-level specification going towards an implementation assisted by a synthesis system. The synthesis system should take away the designer's cumbersome work while the designer takes decisions that require creativity. In contrast to the regular push-button systems some synthesis systems have incorporated a limited degree of user interaction to allow the designer to make decisions. We have chosen a user centred design methodology ([Mid94b], [MMM95]) in which the designer can apply pre-proven behavior preserving transformations ([EMH93],[Raj94]) on a CDFG. The designer selects a transformation and a position where the transformation should be applied. Therefore, it is desirable that the translation process is straightforward. Using a syntax based approach facilitates this. The set of transformations the designer can choose from is quite large and consists of e.g. algebraic transformations like commutativity and distributivity, boolean transformations like De Morgan's law, hierarchy and type transformations. Furthermore many transformations known from compiler design are being used, among which are: constant propagation, dead code elimination, common subexpression elimination and loop transformations. Naturally, there is a set of hardware specific transformations too, e.g. retiming and scheduling. To apply the distributivity transformation (see figure 2) a '+'-node and a ' '-'-node must be selected and the transformation has to be chosen from a menu. The system will check the transformation's preconditions and execute the transformation if allowed. Since a designer does not want to do everything by hand, tools based upon transformations can be constructed (e.g. scheduling). Using these transformations and transformation based tools the design is correct by construction, which eliminates the need for simulation cycles to show that the implementation still meets the specification.

Removing the simulation cycles allows the designer to spend more time on the design.

![Distributivity transformation](image)

3. Design Representation

In order to prove transformations in advance our system is based on a control data flow graph with formal semantics[Huk94], SIL ([Klo92],[KMN92]), our CDFG, was designed within Esprit project SPRITE as a co-operation between Philips Research Laboratories, IMEC and the University of Twente. SIL was designed to serve as an intermediate between hardware description languages and silicon compilers. It allows easy translation of applicative descriptions (e.g. Silage) as well as sequential style descriptions (e.g. C). SIL supports hierarchy, conditional execution and has specialised edges for ordering. These so called sequence edges are drawn as dotted edges. Furthermore, complex types can be defined and computations can be specified on types allowing parameterised designs. These features however, are beyond the scope of this paper. To explain the basic features of SIL figure 3 gives a small example which is the syntactical translation of the VHDL statement 'IF b>c THEN a:=a+1 ELSE a:=a-1 END IF'.

SIL is based on a single token flow model, i.e. exactly one token can reside on an edge at a certain moment. A node fires a token if tokens have been received at all inputs (access points) of that node. Looking at figure 3, the solid lines are data flow edges, the dotted lines are sequence edges and the squared nodes are constants. The bullets at the right side of the nodes are conditions. In the case of an open bullet, which is a true condition, a node
operates normally if the value on the condition is true. If execution is denied the node fires a so-called empty token. In the case of a solid bullet the node operates normally if the value on the condition is false and fires an empty token if the condition is true. If a nodes has more than one condition no condition should issue a deny in order to allow the execution of the node. A sequence edge guarantees that the access point the sequence edges starts from fires before the access point the edge points to. When two data flow edges are firing to the same access point, this is called a join. A join takes the last non-empty token and requires an ordering on the incoming data flow edges. Note that the joins in graph ‘A’ and ‘B’ are generated by the VHDL to SIL compiler to keep the compiler simple. Also the assign-nodes (=) are only generated for convenience. The superfluous joins and assign-nodes can be removed easily using transformations.

Example1:

Figure 3. IF b>c THEN a := a+1 ELSE a := a-

The inputs and outputs of the graph can be associated with the inputs and outputs of the system. Alternatively a graph can be instantiated as a node (the grey nodes in figure 3). Each graph has a name which is printed above it (followed by a colon). Therefore, the grey node with the 'A' in it refers to the graph named 'A:'. Then the input nodes of the graph are associated with the input access points of the instantiated node, the output nodes are associated with the output access points. This way hierarchical graphs can be constructed. Recursions are used to model loops. Furthermore, a special node exists to preserve state, the delay node. When a token enters a delay node it fires the token received previously.

4. The translation model

The following paragraphs describe the principles used for translating VHDL to SIL. In section 4.1 it is explained how sequential descriptions can be translated. Section 4.2 deals with the extensions that are necessary to allow translation of full VHDL excluding time constructs i.e. no signal assignments using the keyword AFTER and no WAIT FOR-statement are allowed.

4.1. Sequential descriptions

Prior to a statement the VHDL description is in a certain state, which consists of the values of all variables and signals. The statement changes this state by updating a variable or signal. We use this notion of state to facilitate the translation of a sequential description.

Figure 4a) Graph A: a<=0; a<=a+1
Graph B: a:= 0; a := a + 1
For each variable and signal an input-node and an output-node which are connected by an edge are made. If a variable or signal is updated an assign-node is inserted and a join is created. If a variable is used a connection is made to last inserted assign node (or the input node if there were no prior assign nodes). In VHDL signals are not assigned a value immediately. Therefore, if a signal is used, instead of taking the last inserted assign node the value is taken from the input node. In figure 4 the translation of variables and signals is demonstrated.

These principles are convenient for translating sequential descriptions, like VHDL processes. Using hierarchies nested statements can be translated easily. The 'then' and 'else' part of an if-statement each get a new hierarchy. Even nested loops with exit or next statements do not pose a problem. The exit condition is connected to all statements following the exit statement, allowing execution to be denied.

4.2 Concurrent descriptions

The principles mentioned above can be used to translate a single process. Translating concurrent statements requires the model to be extended because of the event mechanism. In VHDL for each concurrent statement an equivalent concurrent process can be described. Therefore, only concurrent processes need to be considered. Each VHDL process behaves as if there were a loop around it. If a process does not contain a wait-statement it executes without suspending. Only when a wait-statement is encountered execution suspends. If all processes have suspended, signals are updated (Since time constructs were excluded signals are indeed updated). The updating of signals can cause events, which can restart some processes. The restarted processes again can cause new events which can cause processes to restart. These steps will be repeated until no process restarts again. Only then time advances and the inputs of the circuit are read.

The VHDL execution mechanism described above must be mapped onto the signal flow graph. This mapping consists of loops around the processes, the modeling of wait statements and events and the delta mechanism. In the following sections these problems are discussed.

- Since a VHDL process behaves as if there were a loop around it, a loop must appear in the SIL graph. The body of the process is translated as described in the previous section. The loop around the process is realised by creating a new graph with two hierarchical nodes. The first is an instantiation of the process body graph. The second is a recursive instantiation of itself, thus a loop is created (see figure 8).

- When a wait-statement is encountered in VHDL, the process suspends. In SIL this is achieved by connecting a condition to all statements in the process (figure 9). The result of the wait-statement, called Exe, is a boolean variable which is connected to these conditions. If the previous value of the wait-statement was TRUE the new value is set to FALSE (figure 12). All statements succeeding the wait-statement are denied. The result of the wait-statement is also used to control the execution of the loop thus stopping the execution of the process. If the process is restarted all statements prior to the wait-statement are denied too since the value is still FALSE. The first statement to be executed is the wait-statement. If there was an event on at least one of the signals the wait-statement was sensitive for the new value of the wait-statement will be TRUE.

- An event is a change of a signal. Therefore for each signal in VHDL an additional signal is created in the graph. This signal carries the old value (and will be named \( S - \)). To detect an event all that is necessary is to compare the current and the old value of a signal. If they are equal there is no event and the result of the wait-statement is FALSE. If they are not equal an event has occurred and all statements following the wait-statement are executed till a wait-statement is encountered.

- If there are more processes each process is translated as described above. All processes operate concurrently and are started simultaneously. If all processes have suspended, signals must be updated and if events have occurred processes should restart, this is called the VHDL delta mechanism. To model this a loop is placed around all processes. The loop is re-entered if an event has occurred. Updating of the signals is simply a matter of making the proper connections. The new signal is the
signal at the output of a process. The old value is the value of the signal before it entered the process and the previous old value can be discarded. To check whether or not there were events each signal is compared with its old value. If one of the signals has changed all processes will be restarted. Each process will continue execution with the wait-statement that caused it to suspend as described above. If the wait was not sensitive for the event that just occurred the result of the wait-statement is FALSE causing it to suspend the process again. If there are no more events the delta loop will halt.

- Now we have a graph that correctly models the VHDL delta delay and event mechanism. The only thing that is needed is a reinstatiation of the graph in a new time step using new inputs. The rate at which the inputs are sampled should be high enough not to miss any change at the inputs. All variables, signals and wait-results together form the state of the graph and are therefore connected to delay nodes to retain their values.

To demonstrate these principles an example is given. Figure 5 shows a VHDL description. It generates the pattern '0001' repetitively. Figures 6 to 12 give the SIL graphs generated by our tool.

```
PROCESS
  VARIABLE i : natural := 0;
BEGIN
  IF i < 3
    THEN
      WAIT UNTIL clk='1';
      i := i + 1;
      o <= '0';
    ELSE
      i := 0;
      WAIT UNTIL clk='1';
      o <= '1';
    END IF;
END PROCESS;
Figure 5: VHDL process which generates 0001
```

Figure 6 shows the toplevel graph with Clk and \( i \) as inputs and \( o \) as an output. The delay nodes could be viewed as registers. This top graph calls the delta mechanism graph given in figure 7. It contains the recursion that is used to restart all processes (in this case there is only one process). Processes may restart if the updating of the signals causes new events.

Therefore all new values of the signals are compared with the previous values. (The value of \( clk \) before the process is compared to the value of \( clk \) at the output of the process).

Updating of the signals is done by ignoring \( clk \), taking \( clk \) as the new \( clk \) and using \( clk \) at the output of the process as the new \( clk \).

Figure 6: Top Graph

```
Figure 7: Delta mechanism
```

Figure 8 shows the process loop. The process loops until it stops in a wait statement, i.e. \( \text{Exe1 or Exe2} \) is FALSE. If \( \text{Exe1} \) and \( \text{Exe2} \) are both TRUE the process was not halted and the loop must restart. Since signals are not updated in the process loop they are not connected to the outputs of the process but (the node named PR) but to the inputs of the graph.

Figure 9 shows the body of the process. It contains the if statement. The then and the else.

Figure 8: Process loop
When Ex1 is FALSE the constant TRUE will be selected in stead of the result of 'i<3' due to the deny condition (solid bullet). Thus the if part is allowed to execute starting with the Wait- 

statement becomes TRUE it allows execution of the statement o<=1.

Then:


Figure 8: Process Loop

<table>
<thead>
<tr>
<th>IN</th>
<th>IN</th>
<th>IN</th>
<th>Ck</th>
<th>Ck</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Figure 10 Then part

<table>
<thead>
<tr>
<th>IN</th>
<th>IN</th>
<th>IN</th>
<th>Ck</th>
<th>Ck</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
</tr>
</tbody>
</table>

In Figure 10 the graph of the Then-part is given. It shows the statements i := i + 1 and o <= '0'. These statements are controlled by the result of the wait-statement. The wait-statement is translated in figure 12. The else-part in figure 11 shows that the statement i:=0 is controlled by the previous result of the wait-statement. If Exe is False execution will resume at the wait statement. The statement i:=0 will not be executed. If thereafter the result of the wait-

The translation of the wait-statement is given in figure 12. An equal node is used to compare the old value of clk (clk-) with the current value of clk to detect an event. This result is connected to an AND node together with the result of the until-expression. If the previous result of the wait statement was TRUE the process should halt. The wait-statement should not be executed and the result set to FALSE. This is achieved by the deny condition on the AND-node.

The translation mechanisms as explained above allows translation of close to full VHDL. However, the graphs are not suitable for synthesis yet. Each process has its own recursion, another recursion models the delta
mechanism and each signal to which some process is sensitive is duplicated. To reduce the graphs behavior preserving transformations are applied after the translation. Naturally not all VHDL descriptions that can be translated can be automatically reduced to a synthesizable graph. However we believe that our method offers a way to handle a larger subset than current high-level synthesis tools.

Every possible path in the process contains a wait statement => process does not loop

Where a path is defined as a way to go through the process description from the start to the end. No conclusion can be drawn for a process with a while statement which contains a wait statement as given in figure 13. We can prove that such a process does not loop by proving that the while-statement is executed at least once. In figure 14 the while-statement is unfolded once. In figure 15 constant propagation is used to show that the condition of the if-statement is always true. These types of transformations can be performed on a graph easily. The transformations on the CDFG’s have been proven to be behavior preserving.

```vhdl
i := 0;
WHILE i < 9 DO
  WAIT UNTIL Clk = '1'
  i := i + 1;
END;
```

**Figure 13**

```vhdl
i := 0;
IF i < 9 THEN
  WAIT UNTIL Clk = '1'
  i := i + 1;
END IF;
WHILE i < 9 DO
  WAIT UNTIL Clk = '1'
  i := i + 1;
END;
```

**Figure 14**

```vhdl
i := 0;
  WAIT UNTIL Clk = '1'
  i := 1;
END IF;
WHILE i < 9 DO
  WAIT UNTIL Clk = '1'
  i := i + 1;
END;
```

**Figure 15**

To remove the process loop recursion in SIL we need to show that one of the Exe variables has become FALSE. In that case the recursive call will not be made and the recursion can be removed. Figure 16 gives a part of the graph of figure 8 (the process loop recursion). It shows

---

1 Note that the following process does not loop
a <= not a
wait on a;
(It does loop at the delta level)
that the inputs Exe1 and Exe2 can be transformed to constants. Exe1 and Exe2 must both be true otherwise the PRL graph would not be started. Then the loop is unfolded once as demonstrated in figure 17.

Now the constants can be moved into the second PR, which will become PR'. The first PR graph will remain unchanged. Some conditions can be removed and the constants (Exe1 and Exe2) can be propagated into the then-graph and into the else-graph. In figure 10 we can see that we can propagate the constant into the wait-graph of figure 12. In figure 12 the constant TRUE on the condition eliminates the entire graph leaving only one constant FALSE. Thereafter the statement ‘i:=i+1’ and ‘0<=’ can both be removed since the conditions are always FALSE. The graph of the else-part (figure 11) can be reduced in the same way. Only the statement ‘i:=0’ remains. Now we can

move the constants out of the then-graph and the else-graph back to the process-graph (PR). The result is shown in figure 18.

Now we need to show that Exe1 and Exe2 will never be TRUE simultaneously. Looking at figure 18 we can see that this is indeed the case. All steps performed above are done using transformations which have been proven to be behavior preserving. In figure 17 the graph PRL will never be executed. Exe1 and Exe2 are never both TRUE therefore the result of the bottom AND-node in figure 17 is always FALSE. Thus removing the recursion.

5.2. Removing the Delta-Mechanism

If all process loops have been removed the delta-mechanism recursion should be removed. The delta recursion is restarted when sensitive signals have changed. In this example only Clk is a sensitive signal. Clk is never updated. In the SII graph of figure 7 both inputs of the EQ-node (equal-node) are in fact connected to the same source. An equal-node with both inputs connected to the same source can be transformed into a constant TRUE. Thereafter, it is not difficult to see that the delta-recursion can be removed.
In the general case removing the delta-recursion is more difficult. The delta recursion must be unfolded until it is possible to show that there are no more signals that can change. Thereafter, the redundant computations caused by unfolding the delta-recursion have to be removed.

Conclusions

We have shown that it is feasible to translate VHDL on a per statement basis to a control data flow graph. This syntax-based approach offers a way to handle a larger subset of VHDL in high-level synthesis. After the translation behavior preserving transformations are used to reduce the graph till it can be used for synthesis. We have shown that the process recursions can be removed. More research is needed to prove in which cases the delta-recursion can be removed. Thereafter, redundant computations need to be removed.

A VHDL to SIL compiler and a graphical tool were implemented. Currently we are working on the implementation of scripts of transformations that can ‘clean-up’ the graphs automatically.

References


