Formal Extraction of Memorizing Elements for Sequential VHDL Synthesis

Ludovic Jacomme, Frédéric Pétrot and Rajesh K. Bawa
Département ASIM, LIP6, Université Pierre et Marie Curie
4 place Jussieu, F-75252 Paris Cedex 05, France
{Ludovic.Jacomme, Frederic.Petrot, Rajesh.Bawa}@lip6.fr

Abstract

In this paper we present a method for latches and flip flops recognition within VHDL descriptions of hardware systems. Due to the simulation based semantics of VHDL, the existing synthesis tools rely on explicit templates to guarantee memorizing element inference. The approach proposed here is based on a formal representation of VHDL in terms of Interpreted Petri Nets (IPN). After the compilation of the description, a Petri Net preserving the simulation semantic is build. In order to simplify the formal recognition of the memorizing elements appearing in the description, the Petri net is reduced to a unique minimal form. Ultimately a set of equations can be extracted, and a formal analysis is performed on all cyclic VHDL symbol assignments. This methodology has been implemented and is illustrated on a representative set of simple VHDL descriptions.

1. Introduction

VHDL [1] semantic explicitly relies on an underlying event-driven simulation mechanism. Existing synthesis tools such as the Synopsys VHDL Compiler [2] or Cadence Synergy [3] are able to treat sophisticated VHDL descriptions. In order to overcome the complexity of the simulation semantic, they all impose their own description style that allow to syntactically make the correspondence between VHDL constructs and hardware elements such as flip flops, latches or tristates. In our approach, as opposed to the pattern matching one, we use a formal model based on Interpreted Petri Nets (IPN) [9] to extract memorizing elements.

We propose the use of an intermediate, synthesis oriented, formal model whose goal is specifically to alleviate the constraints on the programming style. We will also guarantee that the behavioral simulation is identical before and after synthesis. In this paper, we present our method for memorizing elements recognition, which is a part of the synthesis processus. Section 2 presents our synthesis oriented formal model based on Petri Nets. Section 3 deals with some of the main aspects of the memorizing elements recognition after the compilation into our formal model and Section 4 concludes with experimental results and future prospects.

2. VHDL compilation on Petri Nets

Each VHDL process can be seen as a synchronized automaton, where changing the state induces some transformations of internal variable's values and signal's drivers. Thus a process can be easily represented as a Interpreted Petri Net (IPN) [7, 8] Its translation should preserve the simulation semantic, the intermediate information relevant to synthesis, the execution's sequentiality of the process body and also the vectorized symbols.

2.1. Presentation of the Petri Net Formalism

The Petri Net is made of a control part and a data part. The control part represents each VHDL processes and their synchronization mechanisms. The data part represents the informations concerning all the symbols manipulated in the VHDL description such as constants, variables or signals.

2.1.1. The VHDL data part

Each VHDL variables and signals to be synthesized is associated to an element of a set $Data = \{S, V, C\}$ representing its attributes.

S is the set of all signals s defined in the description. Each element s is a triplet $(s_{eff}, s_{evt}, s_{drv})$, and defined as follows : s_{eff} the effective value of s, s_{evt} its event attribute and s_{drv} its driving value.

V is the set of all variables v defined in process p. Each element v is defined by $v = v^p_{val}$, where v^p_{val} represents the current value of v.

C is the set of all constants c defined in the description. Each element c is defined by $c = c_{val}$, where c_{val} is the constant value of c.

2.1.2. The control part

The control part is an Interpreted Petri Net representing the sequential instructions of each process. We distinct two type of transitions : T_{exe} associated to the *EXECUTE* phase of the VHDL simulation cycle, and T_{res} corresponding to the *RESUME* phase.

We associate a list of ordered sequential assignments to the T_{exe} transitions and eventually a firing condition, called *GUARD* condition. We assume also that given one VHDL symbol, only one assignment to this symbol is permitted in a transition. The T_{res} transitions correspond to the implicit or explicit wait statements of each process. They are used as barrier synchronization, first to stop the process when it reaches one and second to wake it up on signal's events occurrences. The firing of a T_{res} transition is possible only if an event has occurred on its sensitivity list.

We note $\mathcal{B}$ the boolean set, and $\mathcal{B}_F$ the set of all boolean functions with parameters in is the power set of $Data$ noted $\mathcal{P}(Data)$. The Petri Net PN_{Data} is defined by : $< P, T, ArcPlace, ArcTrans, G, Trf, Asg, M0 >$ where

- P is the set of all places.
- T is a set of all transitions, $T = T_{exe} \cup T_{res}$ such that $T_{exe} \cap T_{res} = \varnothing$. T_{exe} is the set of all transitions modifying elements of $Data$. T_{res} is the set of all transitions sensible to the appearance of events on elements of S.
- $ArcPlace : P \times T \rightarrow \mathcal{B}$ defined as $ArcPlace(p, t)$ is true iff an arc exists from p to t. A transition t has got a maximum of one predecessor.
- $ArcTrans : T \times P \rightarrow \mathcal{B}$ defined as $ArcTrans(t, p)$ is true iff an arc exists from t to p. A transition t has got a maximum of one successor.
- $G : T \rightarrow \mathcal{B}, G(t) \in \mathcal{B}_F$ is the guard condition of t.
- $Trf : T \rightarrow \mathcal{P}(Data) \times \mathcal{B}_F$, where $Trf(t)$ is an ordered set of couples $\{(d_i, f_i)\}$ with $d_i \in Data$ and $f_i \in \mathcal{B}_F$. $Trf(t)$ indicates the modification applied to a data d while firing the transition t.
- $Asg : T \rightarrow \mathcal{P}(Data)$ is the set of all modified symbols when the transition t is fired.
- M0 is the initial marking of the Petri Net.

2.2. The VHDL compilation to Petri Nets

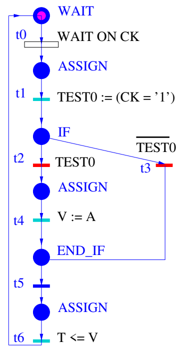
The compilation builds a sub Petri Net for each VHDL process. All sequential statements are individually mapped into an equivalent sub Petri Net. Those resulting sub Petri Nets are connected together by intermediate places. A VHDL description of a flip flop and the corresponding unreduced Petri Net and its $Data$ set are given on Figure 1.

A certain number of reduction techniques [6, 4, 5] allow to obtain a smaller Petri Net, which preserves the underlying VHDL semantic and make the net easier to manage.

The reduction method of the Petri Net can be different depending on the target application. For synthesis, we need to keep all sequences of execution and we also try to obtain

```
ENTITY DREG IS
PORT (
 CK, A : in  BIT;
 T     : out BIT )
END DREG;

ARCHITECTURE BEH
  OF DREG IS
BEGIN
  PROCESS( CK )
    VARIABLE V : BIT;
  BEGIN
    IF (CK = '1')
    THEN
      V := A;
    END IF;
    T <= V;
  END PROCESS;
END BEH;
```




```
Data = {
 (A.eff,A.drv,A.evt),
 (CK.eff,CK.drv,CK.evt),
 (V.val),
 (TEST0.val),
 (T.eff,T.drv,T.evt)}
G(t0) = CK.evt
G(t2) = TEST0.val
G(t3) = NOT TEST0.val
Trf(t1) = {TEST0.val,
          (CK.eff = '1')}
Trf(t4) = {V.val,A.eff}
Trf(t6) = {T.drv,V.val}
Asg(t1) = {TEST0.val}
Asg(t4) = {V.val}
Asg(t6) = {T.drv}
```

Figure 1. VHDL Description of a flip flop

a minimal reduced form, relatively independent of the initial description style.

3. Synthesis of a VHDL process

```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       T     : out BIT )
END DREG;

ARCHITECTURE BEH OF DREG IS
BEGIN
 PROCESS( CK )
  VARIABLE R : BIT;
 BEGIN
  IF ((CK='1') AND CK'EVENT)
  THEN R := A;
  END IF;
  T <= R;
 END PROCESS;
END BEH;
```

```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       T     : out BIT )
END DREG;

ARCHITECTURE BEH OF DREG IS
BEGIN
 PROCESS
 BEGIN
  LOOP
   WAIT ON CK;
   NEXT WHEN (CK/='1');
   T <= A;
  END LOOP;
 END PROCESS;
END BEH;
```

```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       T     : out BIT );
END DREG;

ARCHITECTURE BEH OF DREG IS
BEGIN
 PROCESS( CK )
 BEGIN
  IF(NOT(CK='0') AND CK'EVENT)
  THEN T <= A;
  END IF;
 END PROCESS;
END BEH;
```

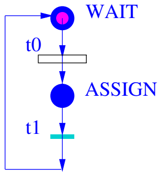
```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       T     : out BIT );
END DREG;

ARCHITECTURE BEH OF DREG IS
BEGIN
 PROCESS( CK )
  VARIABLE R : BIT;
 BEGIN
  IF (CK = '1')
  THEN R := A;
  END IF;
  T <= R;
 END PROCESS;
END BEH;
```

Figure 2. Four equivalent VHDL Descriptions

The synthesis of equivalent single wait processes differs between synthesis tools. For example the four process given in figure 2 are functionally equivalent. They all represent a flipflop as far as VHDL simulation is concerned. Synopsis Synthesis tool [2] is incapable of treating the first three descriptions (Behavioral Compiler Error). However, the fourth one is synthesized as a latch ! Our approach will deduce a flipflop from all four descriptions.

Once the Petri Net is completely reduced, we obtain a set of equations for each variable and signal of the process. The cyclic equation of the driver T_{drv} of the signal T derived from all the four descriptions is illustrated on the Figure 3. Identification of the memorizing elements is done in two distinct steps : the analysis of the sensitivity list and the analysis of the equation itself.




```
Data = {
  (A.eff, A.drv, A.evt),
  (CK.eff, CK.drv, CK.evt),
  (TEST0.val),(T.eff,T.drv,T.evt)}
G(t0) = CK.evt
Trf(t1) = {
   ( TEST0.val, (CK.eff = '1') ),
   ( T.drv, (A.eff AND TEST0.val) OR
            (T.drv AND NOT(TEST0.val)) )}
```

Figure 3. The reduced Petri Net of a flip flop

The analysis of the support of the assigned signal or variable and the sensitivity list of the corresponding process, as shown below, allows us to classify each signal or variable into three distinct classes : un-synthesizable assignment, noncyclic combinatorial assignment and cyclic combinatorial assignment.

we note V^{red} the intermediate variables created during the reduction phase. Given the couple $(d, f_d) \in \mathit{Trf}(t_{asg})$, we note $\Delta_+(f_d)$ the extended support of f_d, where all intermediate variables $v \in V^{red} \mid v \in \Delta(f)$ are replaced by their own extended support $\Delta_+(f_v)$. By construction the extended support $\Delta_+(f)$ of the function f contains only symbols of the two sets S and $V - V^{red}$.

Let $Sens$ the set of all signals s_i that appear in the sensitivity list of the process, we note $\Delta_+^s(f_i) = \Delta_+(f_i) \cap Sens$ and we note $\overline{\Delta}_+^s(f_i) = \Delta_+(f_i) - \Delta_+^s(f_i)$.

An analysis is performed for all useful couples $(d_i, f_i) \in \mathit{Trf}(t_{asg})$ with $d_i \notin V^{red}$:

if $\Delta_+^s(f_i) = \varnothing$, then if $d_i \in \Delta_+(f_i)$ it is un-synthesizable, or if $\Delta_+(f_i) = \varnothing$ it is a constant, or if $\overline{\Delta}_+^s(f_i) = \varnothing$ it is a combinatorial function and otherwise it is un-synthesizable.

if $\overline{\Delta}_+^s(f_i) = \varnothing$ then if $d_i \in \Delta_+(f_i)$ it is a cyclic function and otherwise it is a simple combinatorial function.

if $\overline{\Delta}_+^s(f_i) \neq \varnothing$ then if $d_i \in \Delta_+(f_i)$ it is a cyclic function, and otherwise it is un-synthesizable

At this stage, we have classified each useful VHDL symbol into the two synthesizable categories : cyclic and noncyclic functions. Noncyclic functions are simply synthesized as combinatorial gates. The next step is to analyze the cyclic equations, that will possibly be synthesized as memorizing elements.

3.0.1. Analysis of cyclic equations

A one bit memorizing element can be modeled typically by a simple state graph, with a first state associated to the value one, and a second associated to the value zero. We treat all the symbols one by one and construct the boolean equations on all the arcs of the state transition graph to determine :

- The memorization condition,
- Eventual oscillation condition,
- Set and reset conditions,
- The equation of the data to be memorized.

The computation of these equations is performed using BDDs [10]. Given a couple $(d_i, f_i) \mid d_i \in \Delta_+(f_i)$ and two boolean equations $f_i^0 = f_i(d_i = 0)$ and $f_i^1 = f_i(d_i = 1)$ derived from f_i we have :

- $f_i^{Oscil} = f_i^0 \wedge \overline{f_i^1}$ and $f_i^{Mem} = f_i^0 \vee \overline{f_i^1}$,
- $f_i^{Data} = f_i \downarrow f_i^{Mem}$, where $\downarrow$ is the cofactor function,
- $f_i^{Set} = \{f^s \subseteq f_i^{Mem} \mid (f^s = 1) \Rightarrow (f_i^{Data} = 1)\}$,
- $f_i^{Reset} = \{f^r \subseteq f_i^{Mem} \mid (f^r = 1) \Rightarrow (f_i^{Data} = 0)\}$
- $f_i^{Clock} = f_i^{Mem} - (f_i^{Set} \cup f_i^{Reset})$.

Once the above equations have been obtained, we proceed further in the analysis to determine the type of hardware necessary for the synthesis :

- if $f_i^{Oscil} \neq 0 \rightarrow$ an oscillation is possible (undesirable behavior from the designer point of view).
- if $f_i^{Mem} = 1 \rightarrow$ the couple (d_i, f_i) is a simple combinatory assignment, the analysis is ended.
- if $f_i^{Mem} = 0 \rightarrow$ the symbol d_i is never assigned, it's a constant equal to its elaboration value.
- if $f_i^{Set} \neq 0$ or $f_i^{Reset} \neq 0 \rightarrow$ there is a set or reset condition.

An iteration of the previous step may be necessary to remove intermediate variables appearing in the memorizing condition. A more precise analysis has to be performed on the BDDs equations in order to separate the asynchronous, synchronous set or reset conditions, a write enable condition and the clock's edge type. Those new checks have to follow the constraints given by the sensitivity list of the process.

if $\exists\, s \in S \mid s_{evt}$ appears in the function f_i then if $\exists\, c_i \mid f_i^{Clock} = c_i \wedge f_i^{Enable}$, with $c_i = s_{eff} \wedge s_{evt}$ or $c_i = \overline{s_{eff}} \wedge s_{evt}$ then it is a flipflop, and otherwise it is un-synthesizable.

if $\nexists\, s \in S \mid s_{evt}$ appears in the function f_i then if $\Delta_+(f_i^{Data}) \subseteq Sens$, then it is a latch, and otherwise it is un-synthesizable.

It is important to note that these equations do not depend upon the VHDL description style, since the reduction process achieved a minimal form. The set of equations obtained for the descriptions of Figure 3 are the following :

$$
\begin{aligned}
d_i &= T_{drv} \\
f_i &= (A_{eff} \wedge CK_{eff} \wedge CK_{evt}) \vee (T_{drv} \wedge \overline{(CK_{eff} \wedge CK_{evt})}) \\
f_i^0 &= A_{eff} \wedge CK_{eff} \wedge CK_{evt} \\
f_i^1 &= A_{eff} \vee \overline{CK_{eff}} \vee \overline{CK_{evt}} \\
f_i^{Mem} &= (CK_{eff} \wedge CK_{evt}) \\
f_i^{Oscil} &= f_i^{Set} = f_i^{Reset} = 0 \\
f_i^{Clock} &= (CK_{eff} \wedge CK_{evt}) \\
f_i^{Data} &= A_{eff}
\end{aligned}
$$

From this equation all four forms of VHDL process leads to the synthesis of flipflop.

4. Experimental results

We have implemented our approach and have applied it to a set of 54 simple VHDL descriptions. We have also used two well known commercial synthesis tools : Synopsys Design Compiler and Cadence Synergy. This set of descriptions is composed of multiplexors, latches and flip flops with various combinations of write enable, asynchronous or synchronous set and reset, rising or falling clock edges.

Synthesis tools	Error		Warning		Ok	
Synopsys Compiler	26	48.5%	11	20%	17	31.5 %
Cadence Synergy	34	63%	0	0%	20	37 %
Our approach	0	0%	0	0%	54	100 %

Table 1. The compilation of 54 descriptions

In Table 1, the first column gives the name of the synthesis tool. The next two columns give the number and the percentage of VHDL descriptions that have been rejected with an error message by the synthesis tool during the compilation phase. The next two columns give the number and the percentage of thoses which have been accepted with a warning message, and the last two columns thoses which have been accepted without any message.

Synthesis tools	Error	
Synopsys Compiler	13	46%
Cadence Synergy	0	0%
Our approach	0	0%

Table 2. Experimental results after synthesis of VHDL descriptions

In Table 2, the first column gives the name of the synthesis tool. The next two columns give the number and the percentage of VHDL descriptions that have been accepted during the compilation phases, but that were synthesized incorrectly without any error message by the tool. The results Tables 1 and 2 underline three important points :

- Those two synthesis tools don't impose the same restrictions on the description style. It can be damageable for the designs from a portability point of view.
- The Synopsys Compiler is not able to synthesize properly all the VHDL descriptions accepted during the compilation and analysis phases.
- Our analysis succeed in all cases to recognize the right hardware implantation, whatever the style of description.

5. Conclusion

We have presented an approach which exploits a formal model of VHDL in terms of Interpreted Petri Nets. Based on the reduced Petri Net, we are able to determine if the process can be synthesized. This is done by the analysis of the cyclic and noncyclic data equations attached to the Petri Net, taking into account the sensitivity list of the process. If the process is synthesizable, we apply a formal recognition technique to find out the correct hardware required to implement the sequential behavior of the process.

This approach has been implemented and experimented on a set of 54 simple VHDL descriptions, whose goal is to illustrate many ways to describe the same sequential VHDL behavior, such as latches, flip flops, multiplexors or tristates.

Future use of these techniques include first the multi-wait processes synthesis, where both the formal model and the recognition algorithm are useful, and second, the translation to and from VERILOG as the existing translators make use of syntax based rules instead of taking into account the language semantic.

References

[1] VHDL LRM (1992) IEEE Standard VHDL Language Reference Manual in *IEEE Std 1076-1987*.

[2] Synopsys Inc. (1988-1996) VHDL Compiler v3.5a

[3] Cadence System Inc. (1997) Synergy v3.0

[4] Encrenaz E. (1995) Une Méthode de vérification de propriétés de programmes VHDL basée sur des modèles formels de réseaux de Petri. PhD thesis (in French), Pierre and Marie Curie University, Paris.

[5] Bawa R. K. (1996) An Automated Framework for the Formal Verification and Analysis of Systems Described in VHDL, PhD thesis (in French), Pierre and Marie Curie University, Paris.

[6] Müller J. and Krämer H. (1993) Analysis of Multi-Process VHDL Specifications with a Petri Net Model. in *Proceedings of the Euro-Dac'93*, **474-479**, Germany.

[7] Olcoz S. and Colom M. (1993) A Petri Net Approach for the Analysis of VHDL Descriptions, in *Proceedings of the CHARME.*

[8] Mirkowski J. and Bilinski K. et al. (1996) Petri Net Modelling of VHDL Simulation Cycle for High Level Synthesis Purposes, in *Proceedings of VHDL User Forum in Europe SIG-VHDL Spring'96 working conference*, Dresden.

[9] Murata. A. (1989) Petri Nets: Properties Analysis and Applications, in *Proceedings IEEE, vol. 77 N° 4*, **541-580**, April 1989.

[10] Bryant R. E. (1986) Graph Based Algorithms for Boolean Function Manipulation, in *Proceedings of IEEE Transactions on Computers, Vol C-35*, **677–691**, 1986.