Petri Nets as Intermediate Representation between VHDL and Symbolic Transition Systems

Gert Döhmen
Kuratorium OFFIS, Westerstr. 10-12, D-26121 Oldenburg

Abstract

This paper presents a model for VHDL which easily links to fully automatic formal verification methods and is compositional in that it can model isolated open VHDL architecture bodies. A VHDL architecture body is represented by an interpreted open Petri net that is tailored to be transformed into a finite state transition system. "Open" means that the Petri net behaviour is dependent on inputs from the environment. The net construction allows to characterize the transition system by functional reduced ordered BDDs.

1 Introduction

The ESPRIT project FORMAT will provide a formal verification environment for VHDL. This supports symbolic model checking techniques to prove a VHDL design against properties formulated in an assumption-commitment temporal logic. This paper will characterize the intermediate Petri net model used to derive transition systems from VHDL defined in functional reduced ordered BDDs.

The VHDL standard [IEEE87] gives an informal description of the execution model of VHDL. Although there is recently much effort in setting up different kinds of formal semantics for VHDL the common formal semantics is still missing. We assess different approaches with respect to their suitability for fully automatic verification (model checking).

[vTa89] defines a semantics for VHDL in HOL and [BPS91] axiomatises the semantics of VHDL in the Boyer-Moore logic. While these semantics are appropriate to link VHDL to interactive theorem proving (as e.g. described in [BE93]) they are practically not suitable for fully automatic formal verification methods.

[BSD94] defines a declarative semantics for VHDL with a clocked sequence of global states - one for each unit-time interval. This leads to a very fine grained model which makes model checking expensive.

The approach described in [OC93] uses Coloured Petri Nets as semantical foundation of VHDL. Data aspects of VHDL are modeled with infinite datatypes and thus finite state verification methods cannot be applied to verify data dependent aspects with this model.

The Petri net model described in [MK93] is used to determine access conflicts on global signals. The approach described in this paper is based on the Petri net semantics given in [DJS92,DJS93a,DJS93b]. It modifies the given Petri net semantics in order to restrict on those aspects which shall be verified in the context of the FORMAT project and to support the verification of these aspects most efficiently by model checking. This is achieved through substituting the unit-time synchronizations by maximal-delayed ones and keeping the net deterministic to allow a transition system representation with functional BDDs and thus support functional model checking.

This paper is organized as follows. Chapter 2 gives an informal definition of the predicates which can be used in the temporal logic to specify properties of a VHDL behavioural body. It defines also the currently supported subset of VHDL. Chapter 3 defines the class of nets used to model VHDL. Chapter 4 describes the construction of the state space and nets for specific VHDL constructs. Chapter 5 comprises the conclusions.

2 Visibility and supported VHDL subset

The grade of detail of the Petri net model for VHDL depends on the observables which can be used in requirement specifications. In our approach temporal logic is used as a declarative specification of the behaviour of a VHDL behavioural body as observable at its interface. The interface of such a behavioural body is defined by its corresponding entity declaration. Within this paper we envisage a restricted VHDL syntax for entity declarations which allows to declare only ports of mode IN or OUT.
A requirement specification can be attached to an entity declaration. It uses as propositions boolean-valued VHDL expression referencing as basic elements only input and output ports from the corresponding entity declaration as well as constants and auxiliary variables. Functions are not allowed. Input ports have to be mapped to inputs of the Petri net which are written by the environment and Output ports have to be mapped to variables containing the effective value of it. Additionally the propositions \( \text{AT\_WAIT} \), \( \text{ENV\_AT\_WAIT} \) and \( \text{ENV\_DEL\_POS} \) can be used. \( \text{AT\_WAIT} \) characterizes that the behavioural body under consideration is willing to synchronize with its environment, i.e. all its processes are suspended at wait statements. Section 4.2.5 will show that this proposition is true for a Petri net iff there is one specific control place is marked. \( \text{ENV\_AT\_WAIT} \) characterizes that the environment is willing to synchronize. We will introduce a boolean-valued input \( \text{ENV\_AT\_WAIT} \) which will directly reflect this proposition. \( \text{ENV\_DEL\_POS} \) characterizes that the environment is willing to perform a non-delta-delay step at the next synchronization. This can be used to inhibit that the environment forces only delta-delay steps and that some progress in time will be allowed in order to prove liveness requirements. We will introduce an input \( \text{ENV\_DEL} \) which characterizes the time step the environment wants to perform. \( \text{ENV\_DEL\_POS} \) can then be mapped onto the proposition \( \text{ENV\_DEL} > 0 \).

Our approach supports VHDL with the following restrictions:
- All data types have to be finite.
- No recursion is allowed.
- Generics must be instantiated at model-generation time.
- Delay expressions must be statically computable at model-generation time.
- \( \text{S\_TRANSACTION}, \text{S\_DELAYED}, \text{S\_STABLE}, \text{S\_QUIET}, \text{S\_LAST\_EVENT} \) and \( \text{S\_LAST\_ACTIVE} \) are not supported.
- Resolution functions must be commutative.

### 3 Interpreted Petri Nets for VHDL

The class of Petri nets has been developed in the context of formalizing the semantics of AADL [DD89]. An introduction of these nets can be found in [DJS93a]. These nets are condition/event nets, whose transitions are labeled with state transformations on a state space generated by variable and signal declarations and which places can be annotated with predicates on the state space. The set of places is split into two classes: control places and trigger places. The token game on the control places of the net reflects the control flow of the VHDL processes. Trigger places are annotated with predicates and are typically used to guard transitions. They are marked if their associated predicate is true in the current state. Graphically control places are depicted by double-lined circles and trigger places by single-lined circles.

We use two kinds of arcs pointing from places to transitions: enabling arcs allow to fire their corresponding target transition only if the originating place carries a token. In fact, transitions can fire if all places in their enabling arcs preset carry a token. Firing the transition will not remove the token. consuming arcs model the removal of a token from their originating place if the corresponding target transition fires.

Having both an enabling arc and a consuming arc between a place and a transition will make up a normal arc known from usual condition/event nets.

As pointed out before, transitions can carry a state transformation on the state space. The state space is given by a so-called symbol table containing typed variables and inputs. Inputs can be viewed as variables which may spontaneously change their values. They are used to describe an open system in which these input variables can be written by the environment.

The state transformation of a transition is defined by a list of equations referencing inputs and variables from the symbol table as well as primed variables. They denote the new value of a variable after the transformation. In all equations to one transition a primed variable is allowed to occur only once. Inputs shall never be directly written by the module and need therefore no primed version. A transition without transformation is called a \( \tau \) transition.

Predicates of trigger places are given as conditions referencing variables and inputs from the symbol table.

### 4 Translation of VHDL into Petri Nets

#### 4.1 The State Space for VHDL

A symbol table represents the state space to a Petri net. The integer type and enumeration type constitutes the basic types in such a symbol table for VHDL. Type constructors for arrays can be used to build complex types. A symbol table consists of a list of entries which defines a name and its corresponding object.

The declarations of types, local variables and ports in VHDL induces the following entries in the symbol table:

- An enumeration and integer type can more or less directly be mapped onto a symbol-table entry with an ENUM resp. INTEGER object. Array types are mapped onto a sequence of ARRAY object entries - one for each dimension of the array type.
For each local variable \( *V *\) of type \( V_{\text{type}} \) initialized with \( V_{\text{init}} \) which is defined in a process with label \( \text{lab} \) a corresponding variable \( V_{\text{lab}} \) in the symbol table will be created.

For each input port \( \text{I} \) of type \( I_{\text{type}} \) initialized with \( I_{\text{init}} \) the following entries will be created in the symbol table: A variable \( I_{\text{EFF}} \) carrying the effective value; an input \( I_{\text{LINP}} \) to model the fact that \( I \) is written from the environment; a variable \( I_{\text{LST}} \) which carries the value of \( I \) before the last synchronization.

For each output port \( O \) of type \( O_{\text{type}} \) initialized with \( O_{\text{init}} \) the following entries will be created in the symbol table: A variable \( O_{\text{EFF}} \) carrying the effective value; the type \( O_{\text{V}}.\text{TYPE} \) of an array will contain the projected values for \( O \), which size \( n \) must be sufficient to store in any arising situation all projected values; a variable \( O_{\text{V}} \) carrying the projected values, initialized with an aggregate over \( O_{\text{init}} \); the type \( O_{\text{D}}.\text{TYPE} \) will declare an arraytype containing the delays of the projected values for \( O \); a delay in a cell corresponds to the value in the \( O_{\text{V}} \) cell with the same index. Since the delay for cell \( 0 \) of \( O_{\text{V}} \) is always zero, \( 0 \) don't represent it in \( O_{\text{D}} \). Therefore \( O_{\text{D}}.\text{TYPE} \) starts with cell \( 1 \). Each cell can store a delay of type \( O_{\text{D}}.\text{ELE} \). This type must be sufficient to store all arising delays. A variable \( O_{\text{D}} \) of type \( O_{\text{D}}.\text{TYPE} \). A variable \( O_{\text{M}} \) carrying the index of the cell with the "last" relevant projected value; it is initialized with zero. A cell is \( \text{valid} \) if its index is smaller or equal to \( O_{\text{M}} \). A boolean variable \( O_{\text{A}} \) indicating that a delta-delay signal assignment has occurred on \( O \). It is initialized with \( \text{false} \).

The symbol table contains always the following entries: A boolean-valued input "ENV_{\text{AT\_WAIT}}"; this models directly the proposition \( \text{ENV\_AT\_WAIT} \). An input "ENV_{\text{DEL}}" of an \( \text{INTEGER} \) subtype. The proposition \( \text{ENV\_DEL\_POS} \) is modeled by the boolean expression \( \text{ENV\_DEL} > 0 \); the type of \( \text{ENV\_DEL} \) must be sufficient to handle all delays. A variable "DEL" of an \( \text{INTEGER} \) subtype; this will be used to compute the maximal proceeding time for a synchronization and signal-update step. DEL uses the same type as \( \text{ENV\_DEL} \).

### 4.2 The Petri nets for VHDL

The net construction is done in a compositional way. We attach subnets to all sequential statements, compose these subnets to a subnet for the process where these sequential statements arise and finally compose the subnets for all processes in a behavioural body together with a global net to construct the final net for this behavioural body.

In this paper we will concentrate on the modelling of signal updates and synchronization. These are the most critical aspects to be modeled in Petri nets with respect to the quality of the resulting transition system. We show the nets for the variable assignment, signal assignment and wait statement. It should be clear from [DJS93a] that all other sequential statements can also be modeled as subnets.

Conditions to trigger places are given as boolean expressions in a VHDL style. They can reference variables and inputs from the symbol table. State transformations to transitions are written in a style very similar to variable assignments in VHDL. The primed variable \( V' \) denotes the value of variable \( V \) after the transformation has occurred. A typical transformation is \( V' := V_1 + V_2 \) stating that the new value of \( V \) will be the sum of \( V_1 \) and \( V_2 \). On the right-hand side of such a transformation we allow the construct

\[ \text{if } \langle \text{select-expression} \rangle \]

\[ \text{then } \langle \text{then-expression} \rangle \text{ else } \langle \text{else-expression} \rangle \text{ fi } \]

which selects depending on the \( \text{select} \) either the \( \text{then} \) or the \( \text{else} \) expression. This is not the if-statement from VHDL.

Whenever a condition or transformation in the net is build from single variables, signals or expressions in VHDL all references to variables or input ports have to be replaced by their corresponding objects in the symbol table. We use a function \( \text{SymbMap} \) to denote this replacement. It is defined as follows:

- For a variable \( V \), \( \text{SymbMap}(V) \) denotes the associated variable \( V.\text{process-label} \) in the symbol table.
- For an input port \( P \), \( \text{SymbMap}(P) \) denotes the \( P.\text{EFF} \) variable in the symbol table.
- For an expression \( expr \), \( \text{SymbMap}(expr) \) denotes the expression resulting from the application of \( \text{SymbMap} \) to all variables and input ports in \( expr \).
As pointed out before all delay expressions have to be statically determined when the net is being built. We use a function \textit{TimeVal} to denote the evaluation of a time expression. The resulting value from \textit{TimeVal} is always given in the smallest physical unit used in the behavioural body under consideration.

4.2.1 Variable Assignment. A variable assignment "\textit{V} := expr;" occurring in a process with label LAB is translated into a subnet with one transition. This transition \textit{t} is annotated with the transformation \textit{V.LAB'} := \textit{SymbMap}(expr). It describes the assignment of the value resulting from the evaluation of \textit{expr} to the new (primed) version of \textit{V.LAB}. All denotations of variables and input ports have to be mapped to their corresponding symbol table representatives.

4.2.2 Signal Assignment. The Petri net for the signal assignment has to update the waveform corresponding to the written output port \textit{P}. The waveform is given by the variables \textit{P.V}, \textit{P.D} and \textit{P.M}. According to the type of signal assignment different nets will be generated.

The simplest case is a signal assignment with delta delay "\textit{P} := expr;" for which a subnet with one transition \textit{t} will be created. It looks like the one for the variable assignment but its transformation is different. Here the transition \textit{t} is annotated with a transformation described by:

\begin{align*}
\text{P.V}(0) & := \text{SymbMap}(expr) \\
\text{P.M}' & := 0 \\
\text{P.A}' & := \text{true}
\end{align*}

The transition \textit{t} puts the new value characterized by \textit{SymbMap}(expr) into cell 0 of the \textit{P.V} array. It deletes all subsequent values in the waveform by setting \textit{P.M} to 0. The \textit{P.A} flag will be set to indicate the occurrence of a delta-delay signal assignment on \textit{P}.

For the non-delta-delay case with transport delay "\textit{P} := transport expr after time-expr;" the corresponding waveform will be updated as described in the following picture. It describes the situation before and after a signal assignment.

The new value will be placed into the earliest valid cell of the \textit{P.V} array for which the corresponding \textit{P.D} cell contains a time bigger or equal to the value of \textit{time-expr} given by \textit{TimeVal(time-expr)}. If no such cell exists, i.e. all valid cells contain smaller values, it will be placed into the cell at position \textit{P.M} + 1. The \textit{P.D} cell corresponding to the rewritten \textit{P.V} cell will be set to \textit{TimeVal(time-expr)}. \textit{P.M} is set to the index of the rewritten cells in \textit{P.V} and \textit{P.D}.

The net which performs this transformation is shown in the following picture. Transition \textit{t_1} is annotated with the transformation \textit{P.M'} := \textit{P.M} - 1. Transition \textit{t_2} is annotated with the transformation \textit{P.V'(P.M + 1) := SymbMap(expr)}

\begin{align*}
\text{P.D}(P.M + 1) & := \\
\text{TimeVal(time-expr)}
\end{align*}

\textit{P.M} := \text{P.M} + 1

The trigger conditions are:

\begin{align*}
\phi_1 & := (\text{P.M} > 0) \\
\text{and } (\text{P.D(P.M)} & >= \text{TimeVal(time-expr)})
\end{align*}

\textbf{4.2.3 Wait Statement.} The subnet for a wait statement "\textit{wait on P until bool-expr;}" has to model the synchronization of processes and the fact that the process keeps waiting until an event happens on \textit{P} and the until expression becomes true. The trigger conditions are:

\begin{align*}
\phi_1 & := (\text{P.LST } /\!\!/ \text{P.EFF}) \\
\text{and } \text{SymbMap(bool-expr)}
\end{align*}

\phi_2 := \neg \phi_1

All transitions of the net for the wait statement are \textit{τ} transitions (i.e. without transformation). The suspend transition will act as a synchronization point between processes. So-called resume and suspend control places allow the activation and deactivation of a process (cf. the net for a process statement and a behavioural body below). The suspend transition will pass a token to the suspend control place and through this a "global control" will establish that the subsequent resume transitions can only fire after all processes have suspended and the effectives for all ports are updated.

The condition \phi_1 triggers the transition resume_1 which terminates the wait statement. \phi_1 is true iff an event has
occurred on P (described by P.LST /= P.EFF) and the until expression is true. If \( \phi_1 \) evaluates to false \( \phi_2 \) is true and the firing of resume.2 lets the process keep waiting at this wait statement.

### 4.2.4 Processes

The subnets for sequential statements have always exactly one starting and exiting control place. Sequential composition of sequential statements can then be modeled by identifying the exiting place of a subnet with the starting place of the sequentially next subnet.

After all subnets for a process have been glued together in their sequential order, the cyclic behaviour of the processes is modeled by identifying the start place of the first subnet with the exiting place of the last subnet.

For each process one suspend control place is constructed by identifying all suspend places from the different wait subnets. The same is done with the resume control places for waits. Arcs from the resulting resume place to all the resume transition in the subnets for wait statements allow a suspended process only to proceed if this place is marked. In order to control the starting of the process also by the resume place, a \( \tau \) transition is generated in front of the starting place for the process. An arc from the resume place to this \( \tau \) transition lets the process start only if the resume place is marked. A further control place in the preset of the \( \tau \) is initially marked and inhibits "restarting" of the process.

### 4.2.5 Behavioural Body

The net for a behavioural body combines the nets for all processes of the body together with a global part which updates the waveforms and effective of ports. The trigger conditions and transformations attached to transitions are as follows. transi denotes the transformation attached to transition \( t_i \):

- \( \phi_1 = \text{ENV}_\text{AT_WAIT} \)
- trans1: \( \text{DEL}' := \text{ENV}_{\text{DEL}} \)
- \( \phi_2 = (\text{DEL} = 0) \) or \( P_i.A \) or ... or \( P_m.A \)
- trans2: \( \tau \) transition
- \( \phi_3 = \neg \phi_2 \)
- trans3:
  - \( \text{DEL}' := \text{MIN} \ (\text{DEL} , \text{if } P_i.M > 0 \text{ then } P_i.D(1) \text{ fi } , \) ...
  - \( \text{if } P_m.M > 0 \text{ then } P_m.D(1) \text{ fi } ) \)
- trans4:
  - \( P_{out}.D' := \)
  - \( \text{if } (P_{out}.M = 0) \text{ or } (P_{out}.D(1) > \text{DEL}) \)
  - \( \text{then } (P_{out}.D(1) - \text{DEL} , ..., P_{out}.D(n - 1) - \text{DEL} , 0) \text{ fi } \)

In order to keep the net deterministic, the subnets for processes are activated one after the other through their resume places. This is achieved by identifying the suspend place of one process with the resume place of the literally next. When the suspend place of the "last" process is reached, all processes are suspended at a wait statement.

The proposition AT_WAIT (cf. section 2) is true iff this place is marked. Now, if also the environment is ready to synchronize, the exchange of signal values can take place.

1. Transition \( t_1 \) has the AT_WAIT control place \( s_0 \) in its preset and the trigger place \( s_1 \) with the condition "ENV_AT_WAIT" connected by an enabling arc. Thus it can fire iff both places are marked which means that the module under consideration as well as its environment is ready to synchronize. The transformation to \( t_1 \) will load the DEL variable with the ENV.DEL input.
2. Now either \( t_2 \) or \( t_3 \) and \( t_4 \) can fire.
   - (a) \( t_2 \) is enabled iff either the environment forces a delta-delay step with \( \text{ENV}_\text{DEL} = 0 \) and thus \( \text{DEL} = 0 \) or at least one signal assignment with delta delay has
occurred within a process of the behavioural body. \( t_i \) has no transformation.

(b) \( t_i \) is enabled iff a non-delta-delay step has to be performed, \( t_i \) will compute the time for a maximal step which can be performed and assign it to the DEL variable. This time is given as the minimum from DEL and all valid first delay cells of waveforms.

\( t_i \) will fire after \( t_i \) and will update the waveforms according to the proceeding time \( d \) in the DEL variable. \( d \) will be subtracted from all time values in the DELAYS array for waveforms. Note, that since \( d \) has been computed as minimum from all first valid delay cells and the values ofvalid delay cells grow for rising cells, they cannot become negative. If the first cell of such an array becomes zero and is valid, this array, as well as its corresponding VALUES array, will be shifted to the left. The new value of the DELAYS array for \( P_{out} \) is given by an if-then-else expression. The then case constructs the new non-shifted array by an aggregate in which DEL is subtracted from all \( P_{out} \cdot D \) entries. The else case constructs the shifted array by an aggregate starting with cell \( "2" \) of \( P_{out} \cdot D \) and subtracting DEL from all entries. The new value of the VALUES array is constructed in a similar way. \( P_{out} \cdot init \) shall denote the initialization value for \( P_{out} \). The \( .M \) variable for \( P_{out} \) has to be decremented by one iff \( .D \) and \( .V \) are shifted.

3. Transition \( t_i \) will update the \( .EFF \) variables for all ports as well as the \( .LST \) variables for all input ports. It will also reset all \( .A \) variables. For each input port \( P_i \) its \( .LST \) value variable will be rewritten with its \( .EFF \) value variable. Thus \( P_{out} \cdot LST \) will have the value of \( P_{in} \) from the preceding synchronization.

The \( .EFF \) variable will for each input port \( P_{in} \) be rewritten with the \( .INP \) input. This assignment is like a nondeterministic choice for the input value coming from the environment. The \( .EFF \) variable will for each output port \( P_{out} \) be rewritten with cell zero of the \( .V \) array for \( P_{out} \). Since we don't support resolution functions there shall be at most one process writing on \( P_{out} \) and thus at most one waveform for it. For each output port \( P_{out} \), its \( .A \) variable is set to false.

With \( t_i \) the synchronization and signal updates are completed and control is passed to the resume place of the first net for a process.

5 Conclusion

In this paper we have presented a Petri net model for a subset of VHDL tailored for a subsequent transformation into functional reduced ordered BDDs. The Petri net refers in its annotations of places and transitions to a global state space which provides finite datastructures, which e.g. model waveforms of signals in VHDL. It is "open" in the sense that the state space includes inputs which take values from the environment. The net is constructed compositionally from the sequential statements and processes of a VHDL behavioural body.

The transformation from VHDL behavioural bodies into these Petri nets has been implemented in C with the procedural interface from the LEDA VHDL SYSTEM (LVS) [LEDA94]. LVS is a commercial available analyzer for VHDL. The Petri net generator provides the link to transition systems based on reduced ordered BDDs as described in [HP94, PH94].

6 References


[BSD94] Peter T. Breuer, Luis Sánchez Fernández and Carlos Delgado Kloos, Clean formal semantics for VHDL, EDAC '94


