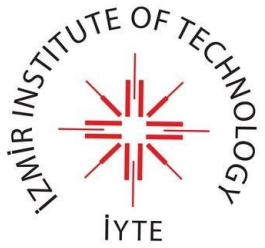


Transforming VHDL to Timed Automata

Tolga Ayav, Tugkan Tuglular, Fevzi Belli
{tolgaayav,tugkantuglular,fevzibelli}@iyte.edu.tr

May 7, 2015

Technical Report
Department of Computer Engineering
İzmir Institute of Technology

35430 Urla İzmir, Turkey. Web: http://compeng.iyte.edu.tr

Technical Report No: IYTE-COMPENG-2015-001
ISSN:
http://arf.iyte.edu.tr/pubs/2015/compeng-2015-001.pdf

This page has been left blank intentionally.

Transforming VHDL to Timed Automata

Tolga Ayav, Tugkan Tuglular, Fevzi Belli

May 7, 2015

Abstract

This report presents the transformation of behavioral VHDL programs to Timed Automata.

1 Background

Synthesizable VHDL is the subset of VHDL, which is used for synthesizing digital circuits. The entire formal grammar definition of VHDL is beyond the scope of this work, yet some of the statements and critical aspects are explained in this context. For complete grammar definition of hardware description languages, see [5] and [3]. Without loss of generality, we restrict our work to the following VHDL syntax for the sake of clarity:

$$
\begin{array}{lcl}
P & ::= & \texttt{entity } N_1 \texttt{ is port}(R) \texttt{ end } N_1; \\
 & & \texttt{architecture } N_2 \textit{ of } N_1 \texttt{ is} \\
 & & [D] \texttt{ begin } C \texttt{ end } N_2; \quad (\textit{Circuit declaration}) \\
C & ::= & s <= e \quad (\textit{Signal assignment}) \\
 & | & s <= e \texttt{ when } b \quad (\textit{Conditional signal assig.}) \\
 & | & \texttt{process}(W) \texttt{ is } [D] \texttt{ begin } S \texttt{ end} \quad (\textit{Process}) \\
 & | & \texttt{for } v \texttt{ in } i_1 \texttt{ to } i_2 \texttt{ generate } C \quad (\textit{Generate}) \\
 & | & \texttt{entity } N \texttt{ port map}(W) \quad (\textit{Comp. instantiation}) \\
 & | & C_1; C_2 \quad (\textit{Parallel composition}) \\
S & ::= & v{:=}e \quad (\textit{Variable assignment}) \\
 & | & s <= e \quad (\textit{Signal assignment}) \\
 & | & a(e_1){:=}e_2 \quad (\textit{Array assignment}) \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ endif} \quad (\textit{conditional}) \\
 & | & \texttt{case } e \texttt{ when } i_1 => S_1 \ldots \\
 & & \texttt{ when } i_n => S_n \texttt{ end case} \quad (\textit{conditional}) \\
 & | & \texttt{for } v \texttt{ in } 0 \texttt{ to } i \\
 & & \texttt{loop } S \texttt{ end loop} \quad (\textit{Iteration}) \\
 & | & S_1; S_2 \quad (\textit{sequencing}) \\
b & ::= & b_1 \odot b_2 \mid \texttt{true} \mid \texttt{false} \\
 & | & v \mid s \mid i \mid \neg b \\
 & | & \texttt{rising\_edge(s)} \\
 & | & \texttt{falling\_edge(s)} \\
e & ::= & i \mid v \mid s \mid a(e) \mid e_1 \odot e_2 \\
 & | & e_1 + e_2 \mid e_1 * e_2 \\
D & ::= & \texttt{variable } v : \texttt{integer } [:= i]; \\
 & | & \texttt{signal } s : \texttt{std\_logic } [:='1' \mid '0']; \\
 & | & \texttt{signal } s : \texttt{std\_logic\_vector} \\
 & & (i_1 \texttt{ to } i_2)[:= i_3]; \\
 & | & D_1; D_2 \\
R & ::= & \texttt{signal } s : \texttt{std\_logic}; \quad (\textit{Port declaration}) \\
 & | & \texttt{signal } s : \texttt{std\_logic\_vector} \\
 & & (i_1 \texttt{ to } i_2); \\
 & | & R_1; R_2
\end{array}
$$

where N is either entity or architecture identifier, v is a variable, s is a signal, a is an array identifier; W is a possibly empty set of signals; i is an integer; and $\odot \in \{\leq, <, =, >, \geq, \texttt{ and }, \texttt{or}, \texttt{xor}\}$. $[D]$ denotes an optional block of signal and variable declarations.

This subset of synthesizable VHDL is quite sufficient to describe circuits used in practice. We give the semantics of the language with brief explanations and small examples below. Please note that by VHDL we mean synthesizable VHDL in the rest of the text.

A VHDL program mainly consists of two parts: *entity* declaration that defines ports of the circuit and *architecture* body that describes what this entity does. The functional program resides in the architecture body. For example, the following VHDL program implements an exclusive-or gate:

```
entity XOR is
   port( x:  in std_logic;
         y:  in std_logic;
         z:  out std_logic);
end XOR;
architecture body of XOR is
begin
   z <= x xor y;
end body;
```

In general, VHDL has two forms of statements: concurrent and sequential. Concurrent statements take place in the architecture body. A concurrent statement can be one of the followings:

- concurrent signal assignment
- *process* statement
- component instantiation statement
- *generate* statement

In the above exclusive-or program, the statement `z <= x  xor y` is a concurrent signal assignment. Processes are such constructions that they might contain variable definitions, their own signal definitions and a sequential code inside its body. Processes are invoked once initially and then only if any change occurs in any signal defined in the sensitivity list. For example, the following code shows a process definition that computes the *xor* of two signals, x and y, and assigns the result to signal z:

```
process (x, y)
begin
   z <= x xor y;
end process;
```

Note that the signals listed between the parentheses in the first line of the above code, *i.e.,* x and y are the elements of the sensitivity list. When a process is invoked, all the statements in its body, *i.e.,* between *begin* and *end* statements, are sequentially executed and then the process halts. Sequential statements may only appear in processes. A sequential statement can be one of the followings:

- signal assignment
- variable assignment
- branching statements such as *if*, *case* and *loop*.

Here, one should notice the semantic difference between variable and signal assignments. In case where a signal is assigned a new value, the assignment is performed when the process halts. On the other hand, assignments are performed immediately in case of variable assignments. Processes can communicate with each other through shared variables and signals provided that multi-source is not used, *i.e.,* two processes cannot update the same variable or signal.

If and *case* statements are quite similar to many high-level languages both in syntactic and semantic manner, therefore they will not be addressed in this context. *Loop* statements need a special care before the transformation; *i.e.,* they must be unrolled so that they can be turned out to be ordinary sequential statements as follows:

$$\texttt{for } v \texttt{ in } 0 \texttt{ to } i \texttt{ loop } S \texttt{ end loop} = \\ v := 0; S; v := 1; S; \ldots; v = n; S;$$

A component instantiation statement in VHDL is similar to placement of a component in a schematic. Component instantiations can be considered as separate circuits. Generate statement simplifies repetitive code and it is used for multiple instantiations of the same component. Therefore, it can be simply considered as multiple component instantiations.

VHDL allows us to describe the circuits, simulate and moreover rapidly realize them, thanks to the Field Programmable Gate Arrays (FPGA). It allows engineers to implement counters, ALU, finite state machines as well as complex digital functions such as microprocessors and digital signal processors [4].

In order to describe a circuit, the model shown in Fig. 1 can be used. The model is generic enough such that both a complex microprocessor and a simple counter conform

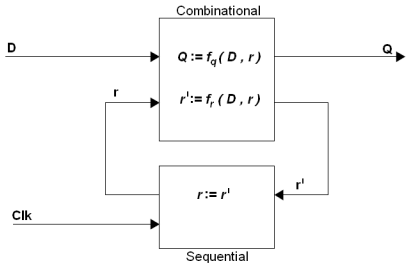


Figure 1: General Structure of Digital Circuits

to the same structure. According to the figure, digital circuits consist of basically a combinational and a sequential part. The sequential part has memory elements such as flip-flops, registers *etc.*, for keeping the state information and they are driven by the clock signal whilst the combinational part generates output signals and decides about the next state depending on the previous state and inputs. In Fig. 1, $f_q(D, r)$ denotes the output function and $f_r(D, r)$ denotes the next-state function both of which depends on the input signals and the previous state in the most general form.

Note that timing statements as to `after 1 us` cannot be synthesized. This is due to the fact that this statement cannot be technically realized on a programmable device since the correct timing can only be satisfied depending on the frequency of an external clock, which is unknown to the chip. On the other hand, timing statements are quite useful for simulation. Therefore, we introduce the following statement in addition to our restricted synthesizable VHDL syntax:

$$s <= e \texttt{ after } t \qquad (\textit{Signal assignment})$$

where t is a strictly positive integer or ∞. We will not dwell upon any further explanations about VHDL since such details are beyond the scope of this paper. For more comprehensive information on hardware description languages, refer to [7] and [5]. Note also that we use VHDL program and circuit interchangeably in the rest of the text, *i.e.,* VHDL programs are considered to be the same with their equivalent circuits.

1.1 Timed Automata

Timed automata is a valuable tool for especially designing real-time systems. In this context, we transform VHDL programs to serve as equivalent timed automata.

Definition 1 *(Timed Automaton). A timed automaton is a tuple* $(Q, q_0, X, \Sigma, \delta, I)$ *where:*

- Q *is a finite set of locations.*
- $q_0 \in Q$ *is the initial location.*
- X *is a finite set of real valued clock variables.*
- Σ *is the set of denoting actions.*

- $\delta \subseteq Q \times 2^C \times \Sigma \times 2^X \times Q$ *is the set of transitions.*
- $I : Q \to 2^C$ *assigns invariants to locations.*

A constraint C is of the form:

$$C ::= z \odot k \quad | \quad z - y \odot k$$

where $z, y \in X$ or $V, k \in \mathbb{N}$ and $\odot \in \{\leq, <, =, >, \geq\}$ and V is a finite set of real valued data variables. A clock valuation is a function $u : X \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^X$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in X$. We will relax the notation by considering guards and invariants as sets of clock valuations, writing $u \in I(q)$ to mean that u satisfies $I(q)$.

Definition 2 *(Semantics of Timed Automaton). Let $(Q, q_0, X, \Sigma, \delta, I)$ be a timed automaton. The semantics is given by a transition system $\langle S, s_0, \to \rangle$ where $S \subseteq L \times \mathbb{R}^X$ is the set of states, $s_0 = (q_0, u_0)$ is the initial state and $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup \Sigma\} \times S$ is the transition relation such that:*

- $(q, u) \xrightarrow{d} (q, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(q)$*, and*
- $(q, u) \xrightarrow{a} (q', u')$ *if* $\exists (q, g, a, r, q') \in \delta : u \in g$*,* $u' = [r \mapsto 0]u$ *and* $u' \in I(q)$*.*

where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock x in X to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $X \setminus r$.

Time may pass only if it satisfies the invariant of the current state. A transition of the automaton may occur if and only if its guard and the invariant of the new state are satisfied. The semantics of the automaton is the set of traces of the associated transition system. Timed automata are often composed into a network of timed automata over a common set of clocks and actions, consisting of n timed automata. The automaton used by UPPAAL is also augmented with a useful transition label, *i.e.,* synchronization channels and data variables in various types. For instance, in case of declaring a synchronization channel a, when a transition labeled with $a!$ occurs, the complementary transition labeled with $a?$ occurs simultaneously. Therefore, we augment the automaton with synchronization channels together with *integer* and *boolean* data variables.

Definition 3 *(Network of Timed Automata). Let $(Q_i, q_i^0, X_i, \Sigma_i, \delta_i, I_i)$ be a network of n timed automata. Let $\bar{q}_0 = (q_1^0, q_2^0, \ldots, q_n^0)$ be the initial location vector. The semantics is defined as the transition system $\langle S, s_0, \to \rangle$, where $S = (Q_1 \times \ldots \times Q_n) \times \mathbb{R}^X$ is the set of states, $s_0 = (\bar{q}_0, u_0)$ is the initial state and $\to \subseteq S \times S$ is the transition relation.*

1.2 Specification Language

Specifications will be expressed in real-time temporal logic TCTL, which extends the the computation tree logic CTL with clock variables.

Definition 4 *(Syntax of TCTL). The formulas φ of TCTL are defined inductively by the grammar*

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \exists\mathcal{U}_I \varphi \mid \varphi \forall\mathcal{U}_I \varphi,$$

where $p \in Pr$ is an atomic proposition and/or clock variables and $I \in \mathcal{I}$ is an interval in the set of intervals $\mathcal{I}$ appearing in φ.

From the above syntax, we can derive the following operators (for further details on the semantics of TCTL and derivation of operators, one may refer to [1, 8] and [2]):

$\exists\Diamond\psi$ (Possibly). There exists a path that property ψ possibly holds.

$\forall\Box\psi$ (Invariantly). Property ψ always holds.

$\exists\Box\psi$ (Potentially always). There exists a path along which property ψ always holds.

$\forall\Diamond\psi$ (Eventually). Property ψ eventually holds.

$\psi \rightsquigarrow \varphi$ (Leads-to). Whenever property ψ holds, property φ eventually holds.

$\psi \rightsquigarrow_{\leq t} \varphi$ (Time-bounded Leads-to). Whenever property ψ holds, property φ eventually holds in at most t time units.

Safety Properties. Safety properties are of the form: "something bad will never happen". For instance, in a model of aircraft, a safety property might be that the altitude must never exceed its maximum value.

Liveness Properties. Liveness properties are of the form: "something will eventually happen", *e.g.*, when pressing the button of the engine start, then eventually the engine should start.

Bounded Liveness Properties. In real-time systems, a liveness property is not sufficient and bounded times response should be investigated. Bounded time liveness property can be expressed with a time-bounded leads-to operator, *i.e.*, $\varphi \rightsquigarrow_{\leq t} \psi$. These properties can be reduced to simple safety properties such that first the model under investigation is extended with a boolean variable b and an additional clock z. The boolean variable b must be initialized to `false`. Whenever φ starts to hold b is set to `true` and the clock z is reset. When ψ commences to hold b is set to `false`. Thus the truth-value of b indicates whether there is an obligation of ψ to hold in the future and z measures the accumulated time since this unfulfilled obligation started. The time-bounded leads-to property $\varphi \rightsquigarrow_{\leq t} \psi$ yields the verification of the safety property $\forall\Box b \Rightarrow z \leq t$. Similarly, we can define $\varphi \rightsquigarrow_{\geq t} \psi$ to express that ψ must hold at least t time units after φ commences to hold.

2 Transforming VHDL Programs to Timed Automata

This section describes the key points of the transformation of VHDL to timed automata.

Please recall that a VHDL program may have concurrent and sequential statements, denoted with C and S respectively. Each concurrent statement can be expressed with a separate automaton.

We define a transformation function $\mathcal{F}$[P] that converts a given program P to timed automata. Transformation process is defined inductively by the the following rules, which must be understood like a case expression in the programming language ML [6]: cases are evaluated from top to bottom, and the transformation rule corresponding to the first pattern that matches the input program is performed. Due to the space limits, we skip some details of this transformation such as the transformation rules for entity and declaration parts. This transformation is relatively straightforward such that all the port definitions, signal and variable definitions are simply transformed to appropriate variables defined in the timed automata.

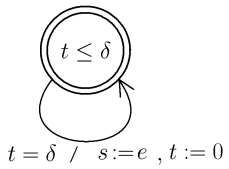
Transformation Rule 1 *(Declarations)*

1. $\mathcal{F}[P] \equiv \mathcal{F}[R] \cup \mathcal{F}[D] \cup \mathcal{F}[C]$
2. $\mathcal{F}\,[\texttt{signal}\ s : \texttt{std\_logic}; R] \equiv \texttt{bool}\ s; \mathcal{F}[R]$
3. $\mathcal{F}[\texttt{signal}\ s : \texttt{std\_logic\_vector}(i_1\ \ \texttt{to}\ \ i_2); R]$
   $\equiv \texttt{bool}\ s[i_1]; \mathcal{F}[R]$
4. $\mathcal{F}\,[\texttt{variable}\ v : \texttt{integer}\ [:= i]; D] \equiv \texttt{int}\ v = i; \mathcal{F}[D]$
5. $\mathcal{F}\,[\texttt{signal}\ s : \texttt{std\_logic}\ [:= b]; D] \equiv \texttt{bool}\ s = b; \mathcal{F}[D]$
6. $\mathcal{F}[\texttt{signal}\ s : \texttt{std\_logic\_vector}(i_1\ \ \texttt{to}\ \ i_2)$
   $[:= i_3]; D] \equiv \texttt{bool}\ s[i_1] = \{i_3 \& (1 \ll i_1),$
   $i_3 \& (1 \ll (i_1 - 1)), \cdots, i_3 \& (1 \ll i_2)\}; \mathcal{F}[D]$

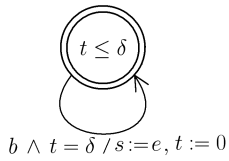
Transformation Rule 2 *(Concurrent statements)*

1.$\mathcal{F}[C_1; C_2] \equiv \mathcal{F}[C_1] \parallel \mathcal{F}[C_2]$

2.$\mathcal{F}[s <= e] \equiv$

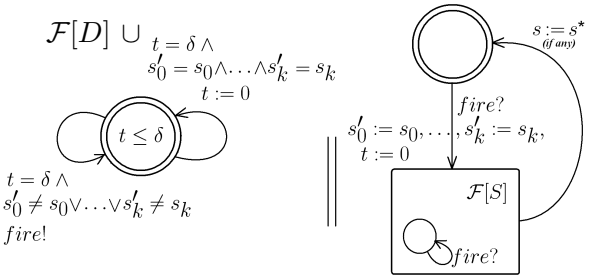


3.$\mathcal{F}[s <= e$ `when` $b] \equiv$



4.$\mathcal{F}[$`for` v `in` i_1 `to` i_2 `generate` $P] \equiv$
$\mathcal{F}[P]_{v:=i_1} \parallel \mathcal{F}[P]_{v:=i_1+1} \parallel \cdots \parallel \mathcal{F}[P]_{v:=i_2}$

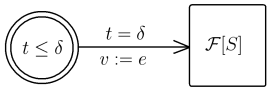
5.$\mathcal{F}[$`process(`$s_0, \ldots, s_k$`) is [D] begin` S `end`$] \equiv$

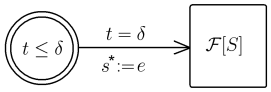


6.$\mathcal{F}[\emptyset] = \mathcal{F}[null] = \emptyset$

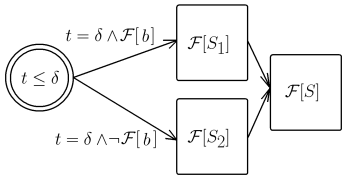
where $C \in \{$ COMPONENT, PROCESS, ASSIGNMENT $(s <= e)$, $\emptyset\}$. Program P is a component that consists of at least one concurrent statement. Components can hierarchically contain other components. Therefore, transformation $\mathcal{F}$ first applies Rule 2.1 to the top-level component P. If the statement is a process, then Rule 2.5 applies. where s'_is are fresh variables, $fire$ is a fresh communication channel and S is a block of sequential statements. S is executed once initially and then only at the times when one of the variables given in the sensitivity list changes. $\mathcal{F}$ produces two automata. The first automaton seen on the left side checks the sensitivity list and decides about triggering the process body S via the synchronization channel $fire$. The second automaton executes S at each trigger.

Transformation Rule 3 *(Sequential statements)*

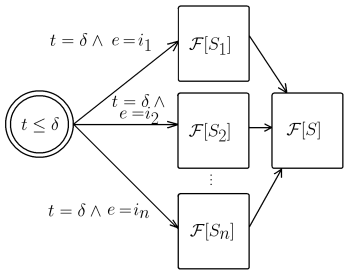
1. $\mathcal{F}[S_1; S_2] \equiv \mathcal{F}[S_1];\mathcal{F}[S_2]$
2. $\mathcal{F}[v := e;S] \equiv$ 

(variable assignment)
3. $\mathcal{F}[s <= e;S] \equiv$ 

(signal assignment)
4. $\mathcal{F}[\texttt{if b then } S_1 \texttt{ else } S_2 \texttt{ ;}S] \equiv$



5. $\mathcal{F}[\texttt{case } e \texttt{ is when } i_1 \texttt{ => } S_1 \ \ldots$
$\texttt{when } i_n \texttt{ => } S_n \texttt{ end case ;}S] \equiv$



6. $\mathcal{F}[\texttt{null}] \equiv \mathcal{F}[\emptyset] \equiv$ 

For condition b and expression e, the following rules apply:

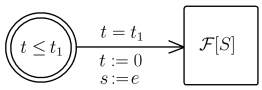
Transformation Rule 4 *(Conditions and Expressions)*

$$1.\ \mathcal{F}[b] = \begin{cases} a' \neq a \ \ \&\& \ \ a = 1 & \textit{if } b = \texttt{rising\_edge(a)} \\ a' \neq a \ \ \&\& \ \ a = 0 & \textit{if } b = \texttt{falling\_edge(a)} \\ b & \textit{otherwise} \end{cases}$$

$$2.\ \mathcal{F}[e] = e$$

where a' is the fresh variable defined during the transformation of the associated process statement. Note that sequential statements must take place in process statements and to use the commands `rising_edge(a)`and `falling_edge(a)` in a process, a must be defined in its sensitivity list, which assures the definition of a'.

Transformation Rule 5 *(Non-synthesizable statement)*

1. $\mathcal{F}[s <= e \texttt{ after } t_1 \ ;S] \ = \ $ 

Transformation rule 5 that includes non-synthesizable statement `after` may seem to incur an antinomy, since we only deal with VHDL in fact. The system under test is the circuit, *i.e.,* VHDL, yet the running circuit can be achieved by providing necessary test signals to the circuit. Note that one may need some non-synthesizable timing statements such as `wait` and `after` to generate clock or other signals.

Timing Information

The time constants, denoted with δ in each rule, can be extracted precisely from the time reports generated by hardware synthesizers. Precise timing knowledge is of paramount importance to check the real-time behaviors of the system, and it depends on the clock frequency and target hardware platform.

References

[1] Mustapha Bourahla and Mohamed Benmohamed. Verification of real-time systems by abstraction of time constraints. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 238.1, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal logic (vol. 1): mathematical foundations and computational aspects.* Oxford University Press, Inc., New York, NY, USA, 1994.

[3] Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee verilog synthesizability. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 41–50, New York, NY, USA, 2008. ACM.

[4] Enoch O. Hwang. *Digital Logic and Microprocessor Design with VHDL.* Brooks / Cole, 2005.

[5] IEEE. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual.* IEEE, 2000.

[6] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[7] Volnei A. Pedroni. *Circuit Design with VHDL.* MIT Press, 2004.

[8] Y. Tachi and S. Yamane. Real-time symbolic model checking for hard real-time systems. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 496, Washington, DC, USA, 1999. IEEE Computer Society.