2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion

Towards Test Case Generation for Synthesizable VHDL Programs Using Model Checker

Tolga Ayav and Tugkan Tuglular
Dept. of Computer Engineering
İzmir Institute of Technology, Turkey
{TolgaAyav, TugkanTuglular}@iyte.edu.tr

Fevzi Belli
Dept. of Computer Science,
Electrical Engineering and Mathematics
University of Paderborn, Germany
Email: belli@upb.de

***Abstract*—VHDL programs are often tested by means of simulations, relying on test benches written intuitively. In this paper, we propose a formal approach to construct test benches from system specification. To consider the real-time properties of VHDL programs, we first transform them to timed automata and then perform model checking against the properties designated from the specification. Counterexamples returned from the model checker serve as a basis of test cases, *i.e.,* they are used to form a test bench. The approach is demonstrated and complemented by a simple case study.**

***Keywords*-Test case generation; model checking; synthesizable VHDL; program transformation; timed automata.**

I. Introduction and Related Work

VHDL is a hardware description language that was originally designed for describing the functionality of circuits [1], [2]. The language is clearly defined and non-ambiguous such that many commercial simulators support almost every defined VHDL construct. A subset of VHDL, called synthesizable VHDL is also used for synthesizing circuits on field programmable gate arrays (FPGAs). Before synthesizing circuits on FPGAs, properties defined in the specification should be checked for conformance. One approach is to use model checkers to verify specified properties. Verification provides guarantees over the entire set of computation paths of a system, but is, in general, very expensive due to the state-space explosion problem. Another approach is testing, in which test cases are generated by some coverage criteria [3] and applied to the system under test (SUT). Test cases are ordered pairs of test inputs and expected test outputs. A test then represents the execution of the SUT using the previously constructed test cases. If the outcome of the execution complies with the expected output, the SUT succeeds the test, otherwise it fails. When the SUT is a VHDL program, it is not feasible to synthesize it to FPGA s and then test it. In such a case, simulation tools, such as Mentor Graphics' ModelSim® and Xilinx's ISE Design Suite®, are used to test VHDL programs. Simulation is relatively inexpensive in terms of execution time, but it only validates the behavior of a system for one particular computation path, which is the test case. In this paper, we present a promising idea for the construction of test benches to test VHDL programs for specified properties using model checkers. To be able to create a test bench, first VHDL programs should be transformed to timed automata. One of the novelties of this work is the transformation rules for VHDL programs. A transformed VHDL program is a network of timed automata, which run concurrently. To be able to run and also test such a network of timed automata, a test bench is required to provide clock and other signals. The test bench can also be a network of timed automata. Our approach proposes to create the test bench using test cases generated by a model checker. To validate our approach, a case study is presented in Section II-A. Model checking uses graph theory and automata theory to automatically verify properties of the SUT, more precisely by means of its state-based model that specifies the system behavior [4]. A model checker visits all reachable states of the model and verifies that the expected system properties, specified as temporal logic formulae, are satisfied over each possible path. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace as a sequence of states [5]. The idea of using traces of counterexamples as test sequences is not new and investigated in various works [5], [6], [7], [8], [9]. Other approaches, such as mutant-based model checking to ensure safety properties [10], exists in the literature. However, those studies did not consider time as part of the specification and thus not as part of the test case.

Note that the approach applies model-based techniques to hardware design, well-known from software engineering. Further features of the approach, which make up its novelty are:

1) Adding formalism into test bench creation for VHDL, which contributes to HW/SW co-design.
2) Definining a concrete method to transform VHDL programs to timed automata.
3) Focusing on *real-time* test case generation, which assists in testing safety critical systems.

The paper is structured as follows: Section II summarizes synthesizable VHDL and timed automata. The proposed approach is explained thoroughly in Section III along with a trivial, widely known example. Finally, Section IV concludes the paper and gives insight into prospective future work.

978-0-7695-4087-0/10 $26.00 © 2010 IEEE
DOI 10.1109/SSIRI-C.2010.22

II. Background

A. *Synthesizable* VHDL

Synthesizable VHDL is used for synthesizing circuits on FPGAs. A synthesizable 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. The full grammar definition of VHDL is beyond the scope of this work, yet some of the statements, which are critical for transformation, are explained using small examples. For full grammar definition of hardware description languages, one may refer to [2] and [11]. 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;
```

From the code transformation's point of view, it could be noticed that 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:

```
compute_xor: 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 construct 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. *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:

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

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.

Synthesizable VHDL allows us to describe the circuits, simulate and physically realize them, thanks to the Field Programmable Gate Arrays (FPGA). FPGA is currently the final point of PLD (Programmable Logic Device) family and commonly used in digital design today. FPGAs are complex programmable logic devices such that Altera's FLEX10K250 has both $250,000$ logic gates and $40,960$ bits RAM on the same chip [12]. FPGAs allow engineers to implement counters, ALU, finite state machines as well as complex digital functions such as microprocessors and digital signal processors [13].

In order to implement a circuit, we can use a generic model to describe the digital circuits. Fig. 1 shows the general structure of digital circuits such that both a complex microprocessor and a simple counter conform to the same structure. According to the figure, digital circuits consist

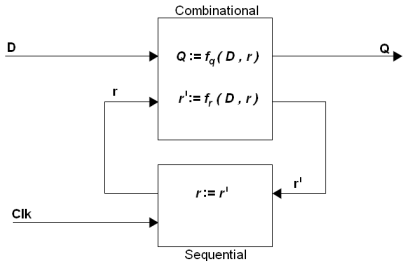


Fig. 1. General Structure of Digital Circuits

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. As an example, an 2-bit down counter will be carried out in the rest of the text. Our 2-bit counter has functionally three inputs D=$\{load, count, d\}$ and one output q. When $load$ is activated, the counter register is loaded with the initial value driven through port d. If $count$ is activated whereas $load$ is inactive, the counter decrements in each clock. All operations are synchronous, *i.e.,* their effects are seen on the output q at rising edges of the clock input. This circuit can be expressed with equations (1) and (2) and its VHDL program is given in Fig. 2.

$$f_r(D,r) = \begin{cases} d & \text{if } load = 1 \\ r & \text{if } load = 0 \wedge count = 0 \\ \max\{r-1,0\} & \text{if } load = 0 \wedge count = 1 \end{cases} \quad (1)$$

$$f_q(D,r) = r \quad (2)$$

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity count2 is
port(
   clk, load, count: in std_logic;
   d: in std_logic_vector(1 downto 0);
   q: out std_logic_vector(1 downto 0));
end count2;

architecture imp of count2 is
signal r, r0 : std_logic_vector(1 downto 0);
begin
  combinational : process(load, count, d, r)
  begin
    if load = 1 then r0 := d;
    elsif count = 1 then r0 := r - 1;
    else r0 := r;
    end if;
    q <= r;
  end process;

  sequential : process(clk)
  begin
    if rising_edge(clk) then r <= r0; end if;
  end process;
end imp;
```

Fig. 2. VHDL program for 2-bit counter

For simulation purposes, we also use the test-bench given in Fig. 3. Please note that the code given in Fig. 2 is synthesizable whereas the testbench's code cannot be synthesized due to the timing statements such as `after 1 us`. This is due to the fact that this statement cannot be realized on a programmable device technically since the correct timing can only be satisfied depending on the frequency of an external clock, which is unknown to the chip. Another example is multiplication statement, *i.e.,* `A*B`. This statement can only be synthesized for certain FPGAs that have ready-to-use multiplication circuits on the chip. Therefore, synthesizable subset of VHDL may show slight

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity testbench is
port(
   q: out std_logic_vector(1 downto 0));
end testbench;

architecture imp of testbench is
 signal clk, count : std_logic;
 signal load : std_logic :='1';
 signal d: std_logic_vector(1 downto 0) := X'02;
 constant Tclk: integer := 10;
 constant Tload: integer := 30;
begin
  clock : process(clk)
  begin
     clk <= not clk after Tclk us;
  end process;

  init : process(load,count)
  begin
     load <= '0' after Tload us;
     count <= '1';
  end process;

  counter: count2 port map(clk,load,count,d,q);
end imp;
```

Fig. 3. Test bench to simulate the 2-bit counter

differences from one FPGA to another, yet such details are beyond the scope of this paper. For further details on FPGA and hardware description languages, please refer to [1] and [2]. Please also note that we use VHDL program and circuit interchangeably in the rest of the text, *i.e.,* synthesizable VHDL programs are considered to be the same with their equivalent circuits.

B. Timed Automata

Timed automata is a valuable tool for especially designing real-time systems. Here, we represent VHDL programs with timed automata. Let X be a finite set of real valued clock variables and V be a finite set of real valued data variables. 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 $\odot \in \{\leq, <, =, >, \geq\}$.

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 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.*
- *$I : Q \rightarrow 2^C$ assigns invariants to locations.*

A clock valuation is a function $u : X \rightarrow \mathbb{R}_{\geq 0}$ 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 abuse 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* $\langle S, s_0, \rightarrow \rangle$ *where* $S \subseteq L \times \mathbb{R}^X$ *is the set of states,* $s_0 = (q_0, u_0)$ *is the initial state and* $\rightarrow \subseteq S \times \{\mathbb{R}_{\geq 0} \cup \Sigma\} \times S$ *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.

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* $\langle S, s_0, \rightarrow \rangle$, *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* $\rightarrow \subseteq S \times S$ *is the transition relation.*

III. Proposed Approach: Test bench Construction for VHDL Programs

VHDL programs are written according to the functional specification and can be executed using a simulator to check whether specified properties are held or not. To check specified properties in our approach, first the model of the system, *i.e.,* VHDL program, is generated using transformation rules introduced in this paper. The obtained model is a timed automata and can be verified using model checker. As explained in Section I, verifying large programs is not feasible due to state-space explosion problem. In our approach, the negation of each specified property is fed to the model checker and its output, as in the form of counterexample, is used to generate a test case. A test bench is constructed by following generated test cases, where the test bench is utilized to test the VHDL program. This approach can be seen in Fig. 4.

A. Transforming VHDL Programs to Timed Automata

The approach of test case generation from an automata based model checker requires modeling VHDL programs, *i.e.,* transforming them to automata. In particular, we focus on timed automata and this section describes the key points of the transformation of VHDL to timed automata.

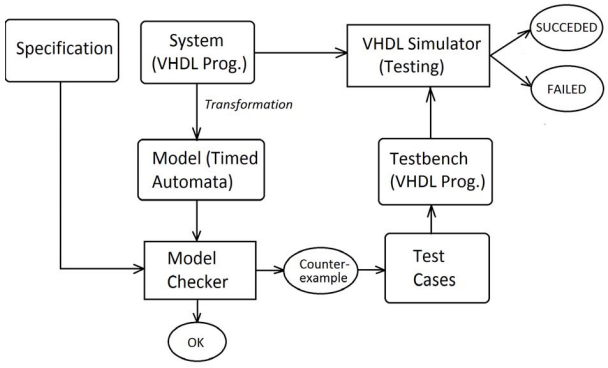


Fig. 4. The approach of circuit testing using model checker

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 $\mathcal{F}[\mathrm{P}]$ 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 [14]: 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. For instance, the following transformation will give some intuition to the reader:

$$\mathcal{F}[\blacklozenge] \;=\; \texttt{int}[0, 255] \quad \texttt{q};$$

where

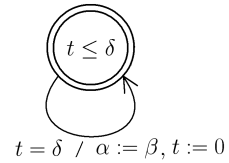
$\blacklozenge$ =

```
entity testbench is
port(
   q: out std_logic_vector(7 downto 0));
end testbench;
```

Transformation Rule 1 *(Concurrent statements)*

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

2. $\mathcal{F}[\alpha <= \beta] \;=$



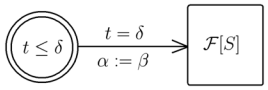
3. $\mathcal{F}[\emptyset] \;=\; \emptyset$

where $C \in \{$ COMPONENT, PROCESS, ASSIGNMENT $(\alpha <= \beta), \emptyset\}$.

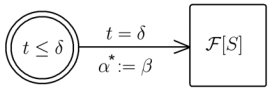
Program P is a component that consists of at least one concurrent statement. Components can hierarchically contain other components. Therefore, transformation $\mathcal{F}$ first applies Rule 1.1 to the top-level component P. If the statement is a process, then Rule 4 applies.

Transformation Rule 2 *(Sequential statements)*

1. $\mathcal{F}[S_1; S_2] = \mathcal{F}[S_1];\mathcal{F}[S_2]$

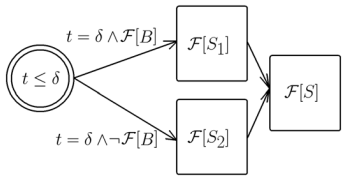
2. $\mathcal{F}[\alpha <= \beta;S] =$ 

(variable assignment)

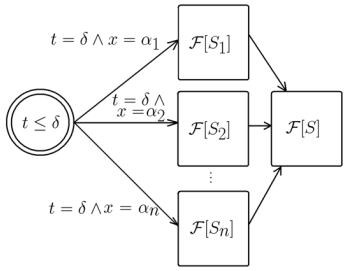
3. $\mathcal{F}[\alpha <= \beta;S] =$ 

(signal assignment)

4. $\mathcal{F}[$`if B then` S_1 `else` S_2 `;`$S] =$



5. $\mathcal{F}[$`case` x `is when` α_1 `=>` S_1 `...`
`when` α_n `=>` S_n `end case ;`$S] =$



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

where $B ::= a \odot b \mid$ `rising_edge(a)` $\mid$ `falling_edge(a)` $\mid \neg$`a`, $\odot \in \{\le, <, =, \neq, >, \ge\}$ and $S \in \{$ `wait, after, if, case, signal assignment, variable assignment, null`$\}$. For condition B, the following rules apply:

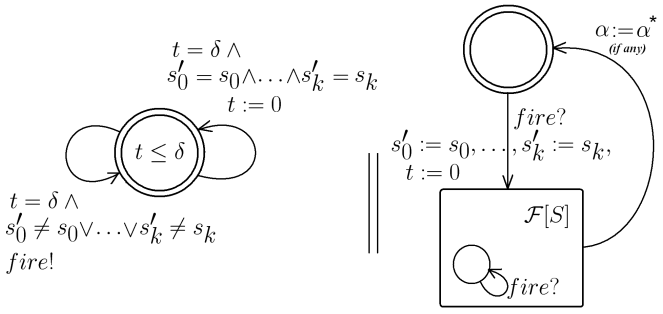
Transformation Rule 3 *(Condition)*

1. $\mathcal{F}[$`rising_edge(a)`$] = a' \neq a \;\&\&\; a = 1$
2. $\mathcal{F}[$`falling_edge(a)`$] = a' \neq a \;\&\&\; a = 0$
3. $\mathcal{F}[B] = B$ *otherwise*

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'.

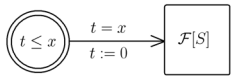
Transformation Rule 4 *(Process statement)*

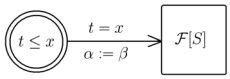
$\mathcal{F}[$`PROCESS`$] = \mathcal{F}[$`process(`$s_0, \ldots, s_k$`) is begin` S `end process`$] =$



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. Note that in order to allow S to execute initially, at least one of the fresh variables s'_is must be assigned an initial value that is different than the current value of s_i.

Transformation Rule 5 *(Non-synthesizable statements)*

1. $\mathcal{F}[$`wait until` $x;S] =$ 

2. $\mathcal{F}[\alpha :=$ β `after` $x;S] =$ 

Transformation rule 5 that includes non-synthesizable statements `wait` and `after` may seem to incur an antinomy, since we only deal with synthesizable VHDL in fact. The system under test is the circuit, *i.e.,* synthesizable VHDL, yet the running circuit can be achieved by providing necessary test signals to the circuit. In our example 2-bit counter given in Fig. 2, we should use a testbench as given in Fig. 3 in order to verify the circuit. This program supplies the counter with the clock and 2-bit data to be loaded into the counter's register. Therefore, verification must be performed on the testbench. Note that one may need some non-synthesizable timing statements such as `wait` and `after` to

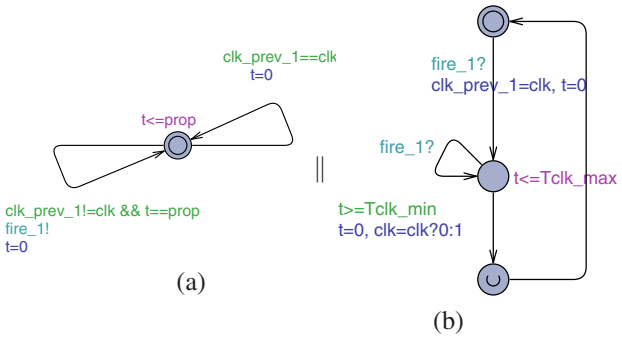


Fig. 5. $\mathcal{F}^{\frac{Tclk\_max}{Tclk\_min}}[\texttt{PROCESS}_{\texttt{clock}}]$. (a) Sensitivity Handler. (b) Process Body.

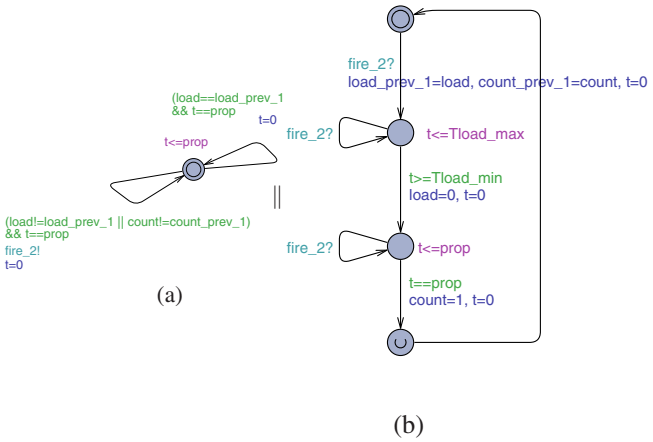


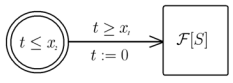
Fig. 6. $\mathcal{F}^{\frac{Tload\_max}{Tload\_min}}[\texttt{PROCESS}_{\texttt{init}}]$. (a) Sensitivity Handler. (b) Process Body.

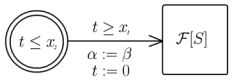
generate clock or other signals in the testbench. The VHDL code of the testbench has three concurrent statements: two processes labeled with "clock" and "init" and one component instantiation labeled with "counter". These three statements can be individually translated into three automata. The component counter, whose code is given in Fig. 2, consists of two processes labeled with "combinational" and "sequential". Consequently, the third automaton can be further divided into two automata. Moreover, each process is transformed into two automata using Rule 4. This means that the whole system can be represented by a network of eight automata as follows:

$$\begin{aligned}\mathcal{F}[\texttt{COMPONENT}_{\texttt{testbench}}] &= \mathcal{F}[\texttt{PROCESS}_{\texttt{clock}}] \parallel \mathcal{F}[\texttt{PROCESS}_{\texttt{init}}] \\ &\quad \parallel \mathcal{F}[\texttt{COMPONENT}_{\texttt{counter}}] \\ &= \mathcal{F}[\texttt{PROCESS}_{\texttt{clock}}] \parallel \mathcal{F}[\texttt{PROCESS}_{\texttt{init}}] \\ &\quad \parallel \mathcal{F}[\texttt{PROCESS}_{\texttt{combinational}}] \parallel \mathcal{F}[\texttt{PROCESS}_{\texttt{sequential}}]\end{aligned}$$

For the VHDL command `wait x`, Rule 5 generates a state that waits for a fixed x units of time. Instead, we can modify Rule 5 such that the waiting time x can be expressed as a range of $[x_1, x_2]$. This allows us to verify the circuit against a given time range of a signal, *e.g.*, the clock. The modified rule can be expressed as follows:

Transformation Rule 5 **(Non-synthesizable statements)*

1. $\mathcal{F}^{x_2}_{x_1}[\texttt{wait until } x;S] \quad =$ 

2. $\mathcal{F}^{x_2}_{x_1}[\alpha := \beta \texttt{ after } x;S] \quad =$ 

Figures 5,6,7 and 8 show the output of transformations $\mathcal{F}[\texttt{PROCESS}_{\texttt{clock}}]$, $\mathcal{F}[\texttt{PROCESS}_{\texttt{init}}]$, $\mathcal{F}[\texttt{PROCESS}_{\texttt{sequential}}]$ and $\mathcal{F}[\texttt{PROCESS}_{\texttt{combinational}}]$ respectively.

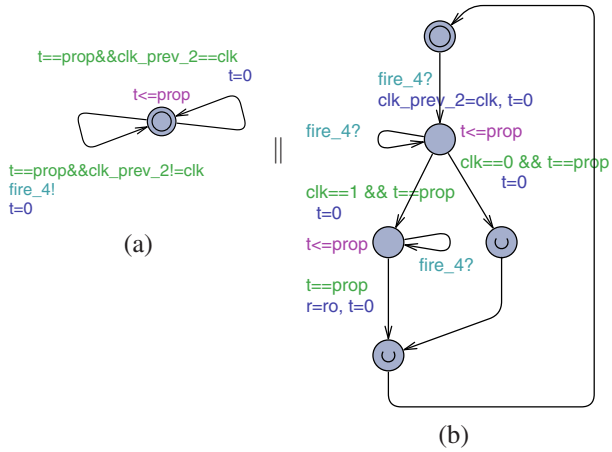


Fig. 7. $\mathcal{F}$ $[\texttt{PROCESS}_{\texttt{sequential}}]$. (a) Sensitivity Handler. (b) Process Body.

B. Test Case Generation Using Model Checker

As a model checker, we use UPPAAL[1]. UPPAAL extends the timed automata with additional features such as bounded integers variables, constants, urgent and committed locations, synchronization channels, *etc.*[15]. The query language of UPPAAL is a subset of CTL (Computation Tree Logic) [16]:

- $E\Diamond\psi$ (Possibly). There exists a path that property ψ eventually holds.
- $A\Box\psi$ (Invariantly). Property ψ always holds.
- $E\Box\psi$ (Potentially always). There exists a path along which property ψ always holds.
- $A\Diamond\psi$ (Eventually). Property ψ eventually holds.
- $\psi \rightsquigarrow \varphi$ (Leads-to). Whenever property ψ holds, property φ eventually holds.

[1]UPPAAL is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University (`http://www.uppaal.com`).

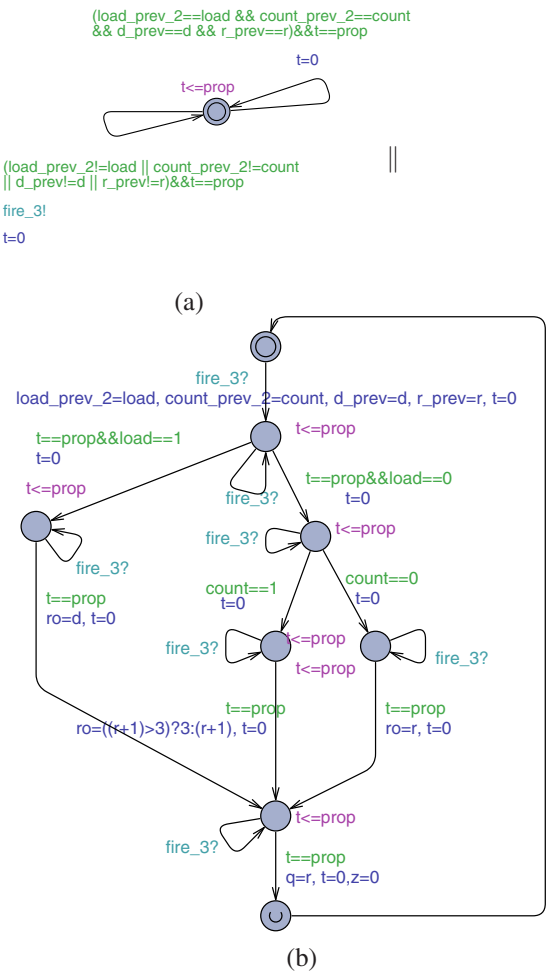


Fig. 8. $\mathcal{F}$ [PROCESS$_{\texttt{combinational}}$]. (a) Sensitivity Handler. (b) Process Body.

$\psi \rightsquigarrow_{\leq t} \varphi$ (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

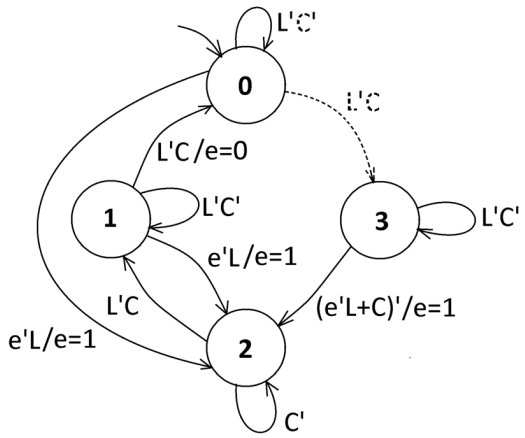


Fig. 9. FSM of the 2-bit down counter

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 $A \,\square\, b \Rightarrow z \leq t$. 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.

In order to be able to enrich the example, we add one more specification to the counter such that once the counter is initialized, *i.e.*, its register is loaded from port d, it cannot be initialized until it reaches to zero. This can easily be implemented by augmenting the register r with an enable bit e and setting and resetting this bit in appropriate places. To simplify the explanation, the timed automata $\mathcal{F}[\mathrm{P}]$ can be reduced to the FSM diagram shown in Fig. 9, where L, C and the transition drawn with the dashed line denote *load*, *count* and a bad transition that must never occur respectively. Assume that the specification of 2-bit down counter contains the following two properties:

$$\begin{aligned} \texttt{P1} \quad &: \quad \texttt{A} \Diamond (\texttt{q} = 0), \\ \texttt{P2} \quad &: \quad (\texttt{q} = 0 \wedge \texttt{load} = 0) \;\; \rightsquigarrow_{\geq \texttt{2Tclk}} \;\; (\texttt{q} = 0). \end{aligned}$$

The first property imposes that 2-bit down counter eventually reaches zero. This is an implicit expectation of any down counter. By negating this property, we obtain the property of down counter never reaches zero. When this property is fed to the model checker, it immediately finds a counterexample with the following trace:

$$\begin{aligned} < \; & [(t,0),(L,1),(C,0),(d,2);(q,2)], \\ & [(t,10),(L,0),(C,1),(d,2);(q,1)], \\ & [(t,20),(L,0),(C,1),(d,2);(q,0)] > \end{aligned}$$

This trace, *i.e.,* test sequence, will be used to test P1. The second property imposes that 2-bit down counter after reaching zero stays at zero unless it is re-initialized. By negating this property, we obtain the property of 2-bit down counter moves to 3 after reaching zero within less than two clock cycles. When this property is fed to the model checker, it immediately finds a counterexample with the following trace:

$$\begin{aligned} < \ & [(t,0),(L,1),(C,0),(d,2);(q,2)], \\ & [(t,10),(L,0),(C,1),(d,2);(q,1)], \\ & [(t,20),(L,0),(C,1),(d,2);(q,0)], \\ & [(t,30),(L,0),(C,1),(d,2);(q,3)] > \end{aligned}$$

This trace will be used to test P2. Due to formal definition of properties, it is straightforward to conclude about the expected output of the test case. This eliminates the problem of test oracle. Moreover, since the test bench will contain all test sequences, full coverage of properties will be achieved within the test suite.

IV. Conclusion and Future Work

In this paper, we proposed an approach towards test case generation for synthesizable VHDL programs. First, VHDL programs are transformed to timed automata, which constitutes the model under consideration, through the introduced transformation rules, and then negation of specified properties written in temporal logic are fed to model checker. Once all the properties are covered, the obtained test suite is used to construct the test bench for the SUT.

The novelty of this approach lies in transformation of VHDL to timed automata and automatic creation of VHDL test benches exploiting software engineering methods. All the methods are presented along with a simple and straightforward yet well-known example for the proof of concept. A more comprehensive application, which is under work, is supposed to rectify the shortcomings of this example, *e.g.,* a simple example is unable to incur a full coverage of practical problems known by software testing community. Moreover, we would like to explore the differences of our approach from other approaches, such as the ones based on Petri nets. Finally, test case generation for timed automata will be further investigated.

References

[1] V. A. Pedroni, *Circuit Design with VHDL.* MIT Press, 2004.

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

[3] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[4] F. Belli and B. Güldali, "A holistic approach to test-driven model checking," in *IEA/AIE'2005: Proceedings of the 18th international conference on Innovations in Applied Artificial Intelligence*. London, UK: Springer-Verlag, 2005, pp. 321–331.

[5] G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: a survey," *Softw. Test., Verif. Reliab.*, vol. 19, no. 3, pp. 215–261, 2009.

[6] P. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM*, 1998, pp. 46–.

[7] G. Devaraj, M. P. E. Heimdahl, and D. Liang, "Coverage-directed test generation with model checkers: Challenges and opportunities," in *COMPSAC (1)*, 2005, pp. 455–462.

[8] A. Engels, L. M. G. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking," in *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 1997, pp. 384–398.

[9] O. Nierstrasz and M. Lemoine, Eds., *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1687. Springer, 1999.

[10] F. Belli, A. Hollmann, and Z. Chen, "Mutant-based model-checking to ensure accessibility and safety aspects of human computer interfaces," in *ICTA*, 2009, pp. 65–74.

[11] J. Gillenwater, G. Malecha, C. Salama, A. Y. Zhu, W. Taha, J. Grundy, and J. 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*. New York, NY, USA: ACM, 2008, pp. 41–50.

[12] ALTERA, *FLEX 10K Embedded Programmable Logic Device Family Data Sheet*, 4th ed., ALTERA, January 2003.

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

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

[15] G. Behrmann, R. David, and K. G. Larsen, "A tutorial on uppaal." Springer, 2004, pp. 200–236.

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