2015 IEEE 39th Annual International Computers, Software & Applications Conference

Model Based Testing of VHDL Programs

Tolga Ayav Tugkan Tuglular Fevzi Belli
Izmir Institute of Technology
Department of Computer Engineering
35430 Urla Izmir, Turkey
{tolgaayav,tugkantuglular,fevzibelli}@iyte.edu.tr

Abstract

VHDL *programs are often validated by means of test benches constructed from formal system specification. To include real-time properties of* VHDL *programs, the proposed approach first transforms them to concurrently running network of timed automata and then performs model checking on properties taken from the specification. Counterexamples generated by the model checker are used to form a test bench. The approach is validated by a case study composed of a nontrivial application running on a microprocessor. As presented, the approach enables testing both hardware and software at once.*

I. Introduction

Circuit functionality is usually defined by VHDL due to its non-ambiguous and clear definition [1]. A subset of VHDL, called synthesizable VHDL, is used for register transfer level (RTL) design, *i.e.*, the first stage of digital integrated circuit design. RTL design is the most difficult part since it is extremely hard to check whether the RTL meets the specifications. Many techniques, such as extensive logic simulation and formal proof models are quite useful; yet none of them is able to offer a satisfying solution to the problem. Another approach is model checking, which provides guarantees over the reachable states. However, model checking comes with its state-space explosion problem. Another approach is to generate test cases following a coverage criteria [2] and apply to the system under test (SUT).

When the SUT is a VHDL program, it is not feasible to test after synthesizing. Therefore, before synthesizing either simulation tools or test benches are used to test VHDL programs [3]. A test bench is, as understood in this paper, a possibly non-synthesizable VHDL code that provides stimulus to the SUT. For this purpose, VHDL program should be transformed to a network of timed automata. To test a concurrently running network of timed automata, a software test bench with clock signal is necessary.

Automatic transformation of VHDL to Timed Automata (TA) is first explored by Nehme [4]. In his thesis, automatic transformation of VHDL to TA is achieved by parsing VHDL code and generating truth table from it and then finite state machine is built from truth table. He designed and implemented so-called VAT tool, which not only transforms VHDL to TA but also is used for verification and validation of embedded systems written in VHDL. In one of the case studies, he used UPPAAL to verify that the model represents the code exactly. However, his study did not comment on test case generation using a model checker for validation purposes.

Model checkers are used to verify properties of a SUT through its state-based model specifying its behavior [5]. A model checker expects properties to be specified as temporal logic formulae, such as "*if state A is reached, then state B must be reached within t time units*". Such properties are checked over all reachable states of the model. If a property can not be satisfied, the model checker tries to produce a counterexample as a sequence of states [6]. Various research [7], [8] utilized counterexamples as test sequences. Another approach used mutant-based model checking to ensure safety properties [9]. Neither of these studies conceived time in the specification. Therefore, time did not appear in the test cases.

Eles et al. [10] worked on the specification of timing constraints in VHDL for high-level synthesis to verify consistency and operation scheduling under timing constraints. To achieve this goal, they developed a notation capturing timing constraints for high-level synthesis with VHDL and an iterative approach to improve VHDL code with back annotation using synthesized times. However, the modeling approach was not TA and testing was not considered. Hessel et al. [11] demonstrated how to au-

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.198

IEEE computer society

tomatically generate efficient real-time conformance test cases from timed automata specifications. The test cases are generated using UPPAAL with optimal execution time. However, test goal generation, test oracle generation and test result checking was not considered in their work.

The problem described in this paper could have been solved also using timed Petri nets for modeling. We decided, however, to model with timed automata as they have already been used in project work. Thus, they were available to no further costs. In future work, timed Petri nets will be considered, especially for comparison of different modeling techniques for VHDL testing. [12].

Note that, in the previous work, we introduced transformation from VHDL to timed automata and demonstrate it with a trivial example [3]. In this paper, we enhance the approach, and to validate it, present a nontrivial case study in Section IV. The approach presented in this paper uses test case generation using model-checking for a circuit and an application running on that curcuit transformed to a software. Major improvements presented in this paper are:

1) The approach has considerably been revised to test both hardware and software at once.
2) To reflect the new approach, the transformation rules have been modified. The previous paper defined 15 rules, the present paper introduces six new rules. Thus, we have now 21 rules in a new, uniform representation format.
3) Therefore, the existing rules have been reformed to get them adapted to the new representation.
4) Section D, "Test Bench Generation from a Test Trace", is an entirely new section that deserves to be extended. We kept it brief due to the page limitation.
5) The case study is new; we validate the approach by a new, non-trivial application.
6) The VHDL code given in Fig.5 is a microprocessor that also includes a software application, which is totally different than the one in our introductory work.

The paper is organized as follows: Section II presents the VHDL syntax used in this study. Section III explains the proposed approach in detail. Section IV validates the approach by a nontrivial case study. Finally, Section V discusses limitations of the proposed approach while Section VI presents conclusion and planned future work.

II. Background

VHDL is a hardware description language and for a complete grammar definition of it, please see [13] and [14]. Without loss of generality, we restrict our work to the following VHDL syntax for the sake of clarity:

$$
\begin{array}{lcll}
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; & (\textit{Circuit declaration}) \\
C & ::= & s <= e & (\textit{Signal assignment}) \\
 & | & s <= e \texttt{ when } b & (\textit{Conditional signal assig.}) \\
 & | & \texttt{process}(W) \texttt{ is } [D] \texttt{ begin } S \texttt{ end} & (\textit{Process}) \\
 & | & \texttt{for } v \texttt{ in } i_1 \texttt{ to } i_2 \texttt{ generate } C & (\textit{Generate}) \\
 & | & \texttt{entity } N \texttt{ port map}(W) & (\textit{Comp. instantiation}) \\
 & | & C_1; C_2 & (\textit{Parallel composition}) \\
 & & & \\
S & ::= & v{:=}e & (\textit{Variable assignment}) \\
 & | & s <= e & (\textit{Signal assignment}) \\
 & | & a(e_1){:=}e_2 & (\textit{Array assignment}) \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ endif} & (\textit{conditional}) \\
 & | & \texttt{case } e \texttt{ when } i_1 => S_1 \ldots & \\
 & & \texttt{ when } i_n => S_n \texttt{ end case} & (\textit{conditional}) \\
 & | & \texttt{for } v \texttt{ in } 0 \texttt{ to } i & \\
 & & \qquad \texttt{loop } S \texttt{ end loop} & (\textit{Iteration}) \\
 & | & S_1;S_2 & (\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} & \\
 & & \quad (i_1 \texttt{ to } i_2)[:= i_3]; & \\
 & | & D_1; D_2 & \\
R & ::= & \texttt{signal } s : \texttt{std_logic}; & (\textit{Port declaration}) \\
 & | & \texttt{signal } s : \texttt{std_logic_vector} & \\
 & & \quad (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 sufficient to describe many circuits used in practice. Note that timing statements like `after 1` μs are not synthesizable. 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 and we need to use them to create test benches in VHDL. Therefore, we introduce the following statement in addition to our restricted synthesizable VHDL syntax in order to be used for testing purpose only:

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

where t is a strictly positive integer or ∞. Timed automata, on the other hand, is a valuable tool for designing particularly real-time systems. In this context, we transform

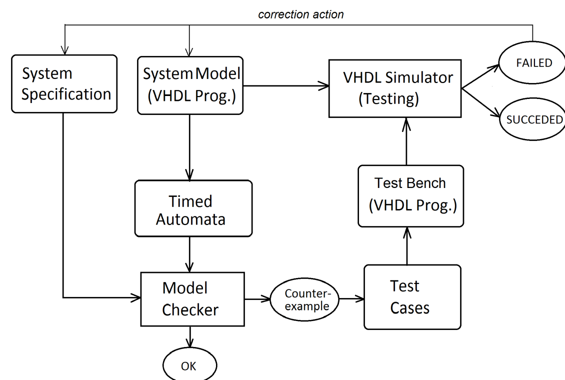

Fig. 1. The approach of circuit testing using model checker

VHDL programs to serve as equivalent timed automata. Time specifications are expressed in real-time temporal logic TCTL, which extends the computation tree logic CTL with clock variables. For comprehensive definition on the semantics of TCTL and derivation of operators, one may refer to [15], [16] and [17].

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. 1.

A. Transforming VHDL Programs to Timed Automata

In the previous work, we defined the transformation function $\mathcal{F}[\mathrm{P}]$ that converts a given program P to timed automata [3]. In this study, we slightly modified the transformation rules depending on the VHDL syntax given in Section II. Due to the page limitation, the transformation rules are given in a technical report [18]. Note that the time constants, denoted with δ in each rule, can be extracted precisely from the time reports generated by hardware synthesizers.

B. Test Case Generation

Our objective guided test case generation algorithm employs a model checker. In the first step, the property declared by the test objective is supplied to the model checker. If the model checker finds a counter-example, the trace found by the model checker is used to define a test case. Then the end location of the counter-example is removed from the model and the model checker is asked to find another counter-example. This process is repeated until the model checker finds no counter-example. This way, the set of property directed test cases are generated. In the second step, the original model is loaded to the model checker and then the property declared by the test objective is negated. The negated property is supplied to the model checker. If the model checker finds a counter-example, the trace found by the model checker is used to define another test case. Then the end location of the counter-example is removed from the model and the model checker is asked to find another counter-example. This process is repeated until the model checker finds no counter-example. This way, the set of negated property directed test cases are generated. The union of property directed test cases and negated property directed test cases constitutes the set of test cases for a test objective. Our objective guided test case generation algorithm should be repeated for each test objective.

Once all the test objectives are consumed and objective guided test cases are generated, the visited locations and/or edges are marked as visited. Then the third step of our high-level test case generation algorithm is executed on the unvisited points. The graph obtained by unvisited points (i.e. locations and/or edges) should be traversed to obtain coverage guided test cases. It is suggested to find a spanning set of entities for some coverage to obtain test cases from a graph [19]. In addition to that, Belli and Budnik [20] suggested minimizing the spanning set to obtain minimum length test cases.

C. Test Bench Generation from a Test Trace

From test sequences, test bench in VHDL can be generated straightforwardly. These test benches are used to simulate the target circuits, *i.e.,* to test them. A test trace or sequence can be given in the following form:

$$< \ [(t,t_1),(\mathbf{i},\mathbf{a_1});(\mathbf{o},\mathbf{b_1})],\ldots,[(t,t_k),(\mathbf{i},\mathbf{a_k});(\mathbf{o},\mathbf{b_k})] >$$

where t is global clock, $\mathbf{i} = [i_1, i_2, \cdots, i_n]$ and $\mathbf{o} = [o_1, o_2, \cdots, o_m]$ are the vectors of input and output signals

respectively. $\mathbf{a_j}$ and $\mathbf{b_j}$ are the associated vectors containing the values ($j \in \{1, 2, \ldots, k\}$). From a given test trace, the test bench in VHDL can be constructed as given in Fig. 2. As seen in Fig. 2, a VHDL test bench consists of

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity testenv is
port(
  i: out type;
end testenv;

architecture imp of testenv is
type state_type is (s0,s1,...,sk);
signal state : state_type := s0;
begin
  Timing : process
  begin
     state <= s1 after t1, ..., sk after tk;
     wait;
  end process;

  Output : process(state)
  begin
     case state is
       when s1 => i <= a1;
       when s2 => i <= a2;
         ⋮
       when sk => i <= ak;
  end process;
end imp;
```

Fig. 2. Test bench in VHDL, which is generated from a test trace

two processes, labeled with "Timing" and "Output". Test sequence has k states and the current state information is stored in register *state*. The former process provides the state changes in appropriate times and the latter one produces the necessary signals in each state.

IV. Case Study

We demonstrate our approach through a well-known seat belt controller application. Our design conducts the soft microprocessor called $\mu Pabs3$. The seat belt controller application is coded in $\mu Pabs3$'s assembly. Thus, our SUT is both the seat belt controller program and the microprocessor itself.

A. Microprocessor $\mu Pabs3$

$\mu Pabs3$ is a fully behavioral VHDL model of a 8-bit small microprocessor designed for educative purposes [21]. It includes almost all essential parts of a general purpose processor. $\mu Pabs3$ has 3 special purpose (IP, IR, FLGS), 25 general purpose (R0-R8 and R16-R31) and 6 general/special purpose registers (R9-R15 : TRIS, P0, TMRH, TMRL, DPTRH, DPTRL, SP). For procedure calls and interrupt mechanism, a stack can also be defined on R31 to R16.

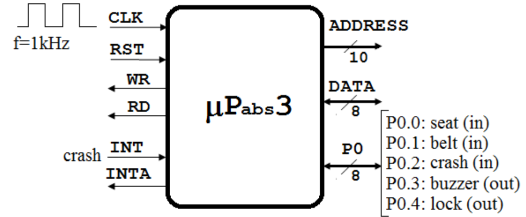

Fig. 3. $\mu Pabs3$ connection diagram for the case study

Execution of 32-bit commands movbi, movwi, jb and jnb take 4 clock cycles (fetch, decode, fetch2 and execute cycles) and the 16-bit rest take 3 clock cycles (fetch, decode and execute cycles).

Due to the space limitation, the VHDL code of $\mu Pabs3$ cannot be given here. The microprocessor's connection diagram for the seat belt application is given in Fig. 3. For further details on $\mu Pabs3$ and its complete VHDL program, refer to [21].

TABLE I. $\mu Pabs3$ commands

Command	Meaning	Cycles
mov dst, src	dst ← src	3
movbi dst, imm8	dst ← imm8	4
movwi dst, imm16	dst,dst+1 ← imm16	4
movx @DPTR, src	(DPTR) ← src	3
movx dst, @DPTR	dst ← (DPTR)	3
add dst, src	dst ← dst + src	3
sub dst, src	dst ← dst - src	3
mul dst, src	dst,src ← dst * src	3
inc reg	reg ← reg + 1	3
dec reg	reg ← reg - 1	3
and dst, src	dst ← dst & src	3
jnz add10	if Z=0 then (SP,SP+1)←IP, SP←SP-2, IP ← add10	3
jtne add10	if TZ=0 then (SP,SP+1)←IP, SP←SP-2, IP ← add10	3
jb reg, imm3, add10	if reg(imm3)=1 then IP ← add10,	4
jnb reg, imm3, add10	if reg(imm3)=0 then IP ← add10,	4
jmp add10	IP ← add10	3
call add10	(SP,SP-1)←IP, SP←SP-2 IP ← add10	3
ret	IP ← (SP+1,SP+2), SP←SP+2	3
banksel b	BNK ← b	3
setb reg, imm3	reg(imm3) ← 1	3
clrb reg, imm3	reg(imm3) ← 0	3
tmr b	TON ← b	3
nop	-	3
halt	-	3

In Table I, dst and src indicate register addresses from R0 to R31. Register addresses can also be used with and indirect addressing prefix @, *e.g.*, mov @R1, R2.

B. Seat Belt Controller

Seat belt controller is widely used in vehicles. In our case study, we implement this controller using $\mu Pabs3$. The FSM diagram shown in Fig. 4 explains how it works. The assembly program implementing the controller can be

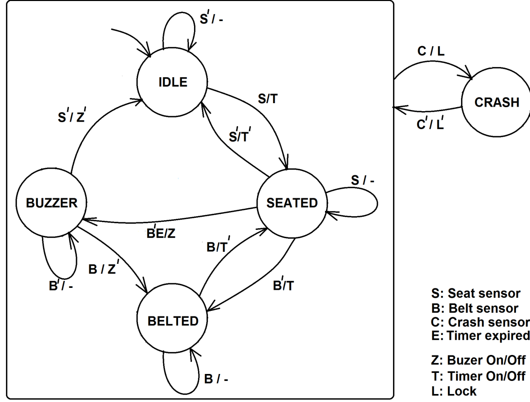

Fig. 4. FSM of the seat-belt controller

written as seen in Fig. 5.

```
              MOV TRIS, 00011000B   ; PORT DIRECTIONS
IDLE:         TMR 0                 ; TIMER OFF
              CLRB P0,4             ; LOCK FREE
              JNB P0,0, IDLE
SEATED:       MOVWI TMR,100
              TMR 1                 ; TIMER ON
S2:           JB P0,1,BELTED
              JNB P0,0,IDLE
              JTNE S2
BUZZER:       SETB P0,3             ; BUZZER ON
              JB P0,1,BELTED
              JNB P0,0,IDLE
              JMP BUZZER
BELTED:       TMR 0
              CLRB P0,3             ; BUZZER OFF
              JNB P0,1,SEATED
              JMP BELTED
EXT_ISR:      SETB P0,4             ; LOCK SEAT BELT
              JB P0,2,EXT_ISR
              RET
```

Fig. 5. Assembly program for the seat belt controller

C. 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.*[22].

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

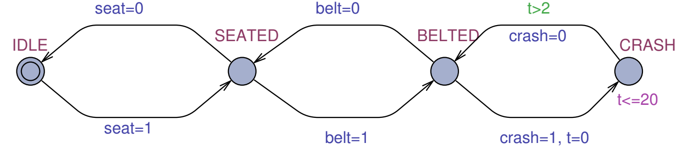

Fig. 6. Test automaton representing the car driver and a crash

Assume that the specification of seat belt controller running on $\mu Pabs3$ contains the following two properties:

$$\texttt{P1} \;:\; \forall \Box \neg(\texttt{seat} \wedge \neg\texttt{belt} \wedge \neg\texttt{buzzer} \wedge \texttt{t} > 100\texttt{T}_{\texttt{clk}})$$

$$\texttt{P2} \;:\; \texttt{crash} \;\leadsto_{\leq 4\texttt{T}_{\texttt{clk}}} \texttt{lock}$$

The first property imposes that seat belt controller never reaches to a state where the driver is seated and not fastened the seat belt and buzzer is not activated even though the timer is expired. The second property tells that if crash happened, lock signal is activated within $4 * T_{clk}$ time units.

The VHDL code of the microprocessor is transformed to its equivalent TA in UPPAAL. Since the processor contains the program memory, the assembly program for the seat belt control application is also included in the resulting TA. In order to run the verifier, we can use the test automaton representing the driver's behavior as given in Fig. 6. According to the test automaton, the driver may seat, fasten the seat belt, then contrarily unfasten the seat belt and leave the car without any invariants and time conditions, *i.e.*, the driver performs each action in the time range of $[0, \infty)$. The model is validated against this behavior.

When these properties are fed to the model checker, it immediately finds counterexamples. From these returned test traces, the test bench is constructed as explained in Section III-C, which may reveal possible error(s) in SUT. In this case study, the test bench helps us to reveal the following coding error in the microprocessor's VHDL program: Fig. 7 shows an excerpt from the VHDL program. Here, the condition expression in line 294 is erroneous. The ">=" operator is corrected as ">" and then Property P1 is shown to be satisfied.

V. Limitations and Threats to Validity

The completeness of transformation rules for given BNF is questionable unless it is proved. It has the highest priority in our future work. Moreover, the scalability of our approach with respect to state explosion and run-time behavior should be discussed and also presented with a larger case study using the operational values obtained from our approach. This requires full automatization of

```
288
289 OUTPUT_LOGIC: process(state)
290 variable mulres: reg16;
291 variable alures: reg8;
292 begin
293
294       if(ton='1' and RF(12)>=X"00") then
295          RF(12) <= RF(12)- X"01";
296       elsif(ton='1' and RF(12)=X"00") then
297          tzero <= '1';
298       end if;
299
300    case state is
301       when s_init =>
302                         IP <= (others=>'0');
303                         for i in 0 to 31 loop
304                               RF(i) <= (others=>'0');
305                         end loop;
306                         RF(15) <= "00011111"; -- SP
```

Fig. 7. Excerpt from the microprocessor's VHDL **code**

testing process. Currently, transformations and checking test results are performed manually. Moreover, the complexity of test computation, which would be a key issue for industrial use of our approach, should be presented for each step of our approach.

VI. Conclusion and Future Work

In this paper, we proposed an approach towards test bench generation for 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 generation of VHDL test bench exploiting software engineering methods. The approach is validated by means of a nontrivial case study. Larger scale applications and their comparisons, which are under work, are supposed to rectify the shortcomings of this example, *e.g.*, one case study 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] 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.

[3] T. Ayav, T. Tuglular, and F. Belli, "Towards test case generation for synthesizable VHDL programs using model checker," in 2nd *Workshop on Model-Based Verification & Validation*, June 9-11 2010.

[4] C. Nehme, "The vat tool, automatic transformation of vhdl to timed automata," Master's thesis, Aeronautics and Astronautics at Massachusetts Institute of Technology, June 2004.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] P. Eles, K. Kuchcinski, Z. Peng, and A. Doboli, "Specification of timing constraints in vhdl for high-level synthesis," 1994.

[11] A. H. Kim, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-optimal real-time test case generation using uppaal," in *In FATES03*. SpringerVerlag, 2003, pp. 114–130.

[12] G. J. Holzmann, *Design and validation of computer protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

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

[14] 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.

[15] M. Bourahla and M. 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*. Washington, DC, USA: IEEE Computer Society, 2003, p. 238.1.

[16] 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*. Washington, DC, USA: IEEE Computer Society, 1999, p. 496.

[17] 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.

[18] T. Ayav, T. Tuglular, and F. Belli, "Transforming VHDL to timed automata," Izmir Institute of Technology, on web: *"http://www.iyte.edu.tr/∼tolgaayav/DCSoC/IYTE-COMPENG-2015-001.pdf"*, Tech. Rep., 2015.

[19] M. Marre and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Transactions on Software Engineering*, vol. 29, pp. 974–984, 2003.

[20] F. Belli and C. J. Budnik, "Minimal spanning set for coverage testing of interactive systems," in *ICTAC*, 2004, pp. 220–234.

[21] T. Ayav, "Lecture notes of ceng311 computer architecture: Design notes of microprocessor $\mu Pabs$," Izmir Institute of Technology, on web: *"http://www.iyte.edu.tr/∼tolgaayav/courses/ceng311/ Design_Notes_of_Microprocessor_uPabs_Ver-08a.pdf"*, Tech. Rep., 2008.

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