This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Kilincceker, O. , Turk, E., Challenger, M., Belli, F., Applying the Ideal Testing Framework to HDL Programs, to appear in Proc. ARCS 2018 – 31st International Conference on Architecture of Computing Systems, 14th Workshop on Dependability and Fault Tolerance (VERFE'18)

Applying the *Ideal* Testing Framework to HDL Programs

Onur Kilincceker
University of Paderborn, Paderborn, Germany.
Mugla Sitki Kocman University, Mugla, Turkey.

Ercument Turk
International Computer Instituite, Ege University, Izmir, Turkey.

Moharram Challenger
International Computer Instituite, Ege University, Izmir, Turkey.

Fevzi Belli
University of Paderborn, Paderborn, Germany.
Izmir Institute of Technology, Izmir, Turkey.

***Abstract*— This paper proposes a framework for testing behavioral model of sequential circuits implemented in Hardware Description Language (HDL). The concept of Ideal Testing is applied for achieving reliability and validity of both positive and negative testing. The HDL program is first modeled by a Finite State Machine (FSM) which is then converted to a Regular Expression (RE). This RE is used to construct test sequences. For positive testing, the original (fault-free) FSM model is used, while for negative testing its mutant model(s) are used to define requirements of ideal testing in conjunction with model-based and code-based mutation testing. A demonstrating example based on a real-life-like Traffic Light Controller (TLC) validates the proposed approach and analyzes its characteristic features.**

Keywords*— *Ideal Testing; Mutation Testing; Test Generation; Behavioral Model; Hardware Description Language; Regular Expression; Traffic Light Controller

I. INTRODUCTION

The complexity of today's Very Large-Scale Integration (VLSI) circuits is increasing based on Moore's Law. So, testing of this type of hardware becomes more critical and overwhelming. There are different abstraction levels of design to deal with the complexity and density of components. Test generation also can be done in various levels of abstraction. For instance, test generation at structural level addresses stuck-at faults in which the main components are gates. We can model a given circuit implemented in Hardware Description Language (HDL) at a lower level such as gate level. Nevertheless, the problem of state space explosion [1] endangers a proper handling over the extraction of a finite state machine (FSM) model at gate level due to the exponential increase of the number of states, based on the number of flip-flops in sequential circuits. However, it is also possible to generate tests in higher level to target stuck-at faults at gate level.

In contrast to traditional testing at gate level, this paper offers an ideal testing methodology at behavioral level using FSM and Regular Expression (RE) to target design errors. The FSM model is converted into a RE that offers abstraction, compactness and conformity to algebraic operations based on Kleene Algebra [2].

Mutation testing is based on the generation of faulty version of system under consideration (namely mutant) considering specific type of fault. For this purpose, a mutation operator is applied into design under test or model to generate mutants. Therefore, it is important to select the mutation operator and type of fault beforehand. This paper uses Model Based Mutation Testing (MBT) technique introduced in [3] in which two types of operators, namely *insertion* and *omission*, are applied on software model. In proposed approach, these operators are applied into FSM model of the Device Under Test (DUT) to generate mutants. Moreover, these mutation operators are also applied into HDL of DUT as being code-based mutation to execute tests in real design, which is different from technique used in [3]. Therefore, the combination of model-based and code-based mutation is used in proposed approach. The mutants are used to pave the way for negative testing.

The current approach makes use of ideal testing concept of Fundamental Test Theorem as introduced by Goodenough/Gerhart, originally for software [4]. An ideal test can detect a specific design error, if the DUT has one. If no error is detected, DUT is free of this design error. This concept is adapted to testing HDL behavioral model.

Briefly, this study attempts to contribute to the literature by adopting an ideal testing framework for testing HDL behavioral model to target design errors. The approach is validated by a traffic light controller example.

Next section presents related work on ideal testing, behavioral level test generation and mutation testing of hardware. Section 3 briefly explains elements of ideal testing. Section 4 explains proposed approach. The proposed approach is demonstrated by an example, a traffic light controller, in Section 5. Section 6 concludes the paper and presents the possible further studies.

II. RELATED WORK

In [4], Goodenough and Gerhart are questioning and examining the basic properties of *ideal* testing such as reliability, validity, and completeness. Howden [5] offers path analysis testing strategy and analysis reliability property of proposed method. Bouge [6] is extending existing formalism of ideal testing to program testing by offering supplementary properties, i.e., bias and acceptability. However, there is no work to our knowledge for application of ideal testing to hardware instead of software.

To test hardware, test generation can be done at various abstraction levels, e.g., structural level (or gate level) or higher levels. At structural level, the process of generating test is called Automatic Test Pattern Generation (ATPG), which strictly uses a netlist (schematic) representation of circuit under test [7]. Nevertheless, it is still open area to develop generation of test in higher levels of abstraction, i.e., register transfer level (RTL) and behavioral level to target manufacturing defects which are represented by fault models, e.g., stuck-at, open, bridging, and delay. Main advantage in higher levels is great reduction in cost of test generation compared to structural level that contains more details than higher level considering complexity of the device.

Test generation or testing at behavioral level is like software testing [8] and requires well-selected fault models to target physical and manufacturing defects. In [9], the authors offer proper fault models and test generation at behavioral level. Moreover, it is reported in [10] that fault models at gate level and behavioral level are correlating. It is also proposing a faster and simpler analysis than in gate level approaches [11].

Mutation testing of hardware is normally addressing to design faults, but it is experimentally shown that they can also detect stuck-at faults in gate level [12,13].

This work proposes a methodology for application of ideal testing to hardware given at behavioral level. To the best of our knowledge, no previous work has been performed on this topic.

III. Ideal Testing

Goodenough/Gerhart's Fundamental Theorem [4] of ideal testing is informally defined as follows:

- Given: DUT; D: Domain of DUT; a Test T $\subseteq$ D; and a test criterium C for test selection (fault definition)
- Test *T* is *successful* if DUT delivers for any t of T a correct output (*t acceptable*),
- Criterium *C* is *reliable* (*consistent*) if all tests, which satisfy C, are successful, OR, if all tests, which satisfy C, are not successful (fail)
- Criterium *C* is *valid* (*effective*) if for each fault in DUT a t of T exists that satisfies C, and reveals the fault
- If a test selection criterium C is reliable and valid, any test T that satisfies C is an *ideal* test.

Fundamental Theorem: If an ideal test succeeds once, it will succeed all tests. Equally, if there is a bug in DUT, an ideal test will reveal it.

To fulfill desired requirements of ideal testing, we propose applying a two-phase procedure including positive and negative testing [14][15] approach, see formal proof in Section 4.2.

- *Positive Testing*: Test sequences are generated from fault-free (original) model
- *Negative Testing*: Test sequences are generated from faulty (mutant) model

In each case, test sequences generated from RE model are applied to both fault-free and faulty DUT to fulfill requirements of positive and negative testing.

In Section 4, where we elaborate the proposed approach, we prove formally how the properties of ideal testing are fulfilled by applying the two-phase testing procedure. In addition, the details of the requirements and test generation procedure from RE model are given in that section.

IV. Proposed Approach

The proposed approach includes two main steps: Testing preparation and Testing/Test-Execution step.

A. Testing Preparation

In *testing preparation* step, see Fig. 1, the DUT is modeled by an FSM and then converted to the corresponding RE by JFLAP tool [16]. Moreover, faults are injected into DUT to acquire *mutants* (faulty versions of the system, implemented, e.g., in HDL). Each mutant can contain one or more errors.

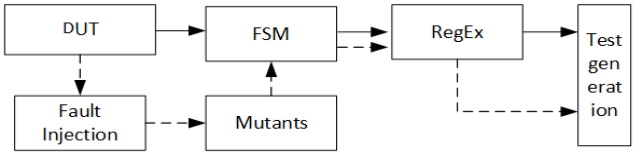

Fig. 1. Testing preparation step

1) Test Generation

A test case is a tuple of input to and expected output and a test sequence (or simply test) is a sequence of test cases. Test generation is process of generation of the test sequences realized based on a RE model.

The RE is basically a sequence of symbols connected by operators. These operators are "." *concatenation*, "+" *selection* (*Union*) and "*" *iteration* (*Kleene's star*). For example;

$$[(a+b)*c] \quad (1)$$

The RE given in (1) contains symbols *a*,*b*, and *c* connected by ".", "+", and "*" operations. The pseudo symbols "[" and "]" represents the start and end of the RE.

REs are parsed into an Abstract Syntax Tree (AST). Test generation from RE requires traversing the AST. As an example, the following test sequences can be generated from given the RE in (1);

$$[],[ac],[bc],[aac],[abc],[bac],[bbc] \quad (2)$$

Due to "*" iteration symbol, length of the test sequences can be infinite, so, it is limited to degree 2 to avoid this scenario. Therefore, the iteration can be repeated one or two times as a constraint. However, test sequences given in (2) are generated under this constraint. If the degree of "*" iteration symbol is represented by n, constraint can be extended to more general form. For example, n = 3 given test sequences in (3) can be generated;

[],[ac],[bc],[aac],[abc],[bac],[bbc],[aaac],[aabc],[abac],[abbc],[baac],[babc],[bbac],[bbbc] (3)

Note that the set of test sequences generated for n = 3 in (3) covers the set of test sequences generated for n = 2 in (2).

B. *Test Execution*

In *test execution* step, see Fig. 2, test sequences generated from RE, defined in previous subsection, are applied to fault-free and faulty devices. As a result, four different testing scenarios can happen;

1. (Positive Testing) Applying test sequences generated from fault-free version to fault-free device
2. (Positive Testing) Applying test sequences generated from fault-free versions to faulty device
3. (Negative Testing) Appling test sequences generated from faulty (mutant) versions to fault-free device
4. (Negative Testing) Applying test sequences generated from faulty (mutant) versions to corresponding faulty device

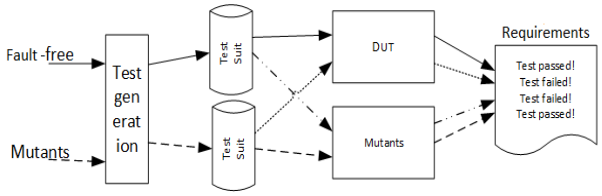

Fig. 2. Testing/Test-Execution step

The test sequences resulting from above mentioned scenarios are selected based on the requirements, which we call them test criteria, defined in TABLE I. Based on the requirements, the test sequences generated from fault-free model are applied to fault-free DUT from which passed test sequences are collected. In addition, the test sequences generated from fault-free model are applied to faulty DUT from which failed test sequences are collected.

TABLE I. REQUIREMENTS FOR DIFFERENT TESTING SCENARIOS

Testing	Fault-Free Version	Faulty Version (Mutant)
(1) Positive Testing	Test passed!	-
(2) Positive Testing	-	Test failed!
(3) Negative Testing	Test failed!	-
(4) Negative Testing	-	Test passed!

In a similar way, the test sequences generated from faulty models are applied to fault-free and the corresponding faulty DUT, from which passed and failed test sequences are collected respectively. In general, an arbitrarily faulty device would often be tested as failed using any other faulty version.

Based on given criteria, resulting from positive and negative testing, ideal testing can be formalized and proven.

1) *Formal proof of proposed two-phase approach for ideal testing*

Assumptions:

For given DUT, Criterium C_1, C_2, C_3, and C_4, defined below, are requirements of the two-phase testing.

In positive testing, test sequence t_i is generated from original model.

- Test sequence t_i passes if it is applied to original DUT (C_1)
- Test sequence t_i fails if it is applied to corresponding mutant DUT (C_2).

In negative testing, test sequence t_j is generated from a mutant model

- Test sequence t_j passes if it is applied to the corresponding mutant DUT (C_3)
- Test sequence t_j fails if it is applied to the original DUT (C_4).

$OK(t_i)$: $(\forall t_i \in T)$ (t_i is acceptable for DUT) and $\neg OK(t_i)$: $(\forall t_i \in T)$ (t_i is unacceptable for DUT).

Proof:

- Successful (T) := $(\forall(t_i) \in T) OK(t_i)$ and Fail(T) := $(\forall(t_i) \in T) \neg OK(t_i)$
  - $((\forall t_i \in T)\ OK(T)\ \ DUT(\forall t_i \in T))$ (t_i acceptable) or $((\forall t_i \in T)\ \neg OK(T)\ \ DUT_m(\forall t_i \in T))$ (t_i unacceptable).
  - $((\forall t_j \in T)\ OK(T)\ DUTm(\forall t_j \in T))$ (t_j acceptable) or $((\forall t_j \in T)\ \neg OK(T)\ \ DUT(\forall t_j \in T))$ (t_j unacceptable).
  - Therefore, t_i and t_j are either succesful or fail.
- Reliable(C) := if all tests, which satisfy C, are successful, OR, if all tests, which satisfy C, are not successful (fail).
  - Positive Testing:$((\forall t_i \in T)\ (Satisfy(t_i,C_1)) \rightarrow$ Successful (t_i) and $(\forall t_i \in T)\ (Satisfy(t_i,C_2)) \rightarrow$ Fail (t_i).
  - Negative Testing:$(\forall t_j \in T)\ ((Satisfy(t_j,C_3)) \rightarrow$ Successful (t_j) and $(\forall t_j \in T)\ ((Satisfy(t_j,C_4)) \rightarrow$ Fail (t_j).
  - Therefore, C_1,C_2,C_3, and C_4 are reliable.
- valid(C) := if for each error in DUT a t of T exists that satisfies C, and reveals the error
  - Criterium C_1: $(\forall t_i \in T)\ (Satisfy(t_i,C_1))\ (DUT(\forall t_i \in T) \rightarrow OK(t_i)$ therefore t_i doesn't reveal the error and C1 isn't valid.
  - Criterium C_2: $(\forall t_i \in T)\ (Satisfy(t_i,C_2))\ (DUT_m(\forall t_i \in T) \rightarrow \neg OK(t_i)$ therefore t_i reveals the error and C_2 is valid.

- Criterium C_3: ($\forall$ $t_j \in T$) (Satisfy(t_j,C_3)) (DUT_m($\forall t_i \in T$) → OK(t_i) therefore t_j doesn't reveal the error and C_3 isn't valid.
- Criterium C_4: ($\forall$ $t_j \in T$) (Satisfy(t_j,C_4)) (DUT($\forall t_j \in T$) → ⌐OK(t_j) therefore ti reveals the error and C_4 is valid.

In short, Criterium C_1 and C_3 are *succesful* and Criterium C_2 and C_4 are *fail*. So, all criteria are *reliable* because all tests, t_i and t_j, satisfying corresponding criterium, are either *succesful* or *fail*. Criterium C_1 and C_3 are *not valid* because tests, satisfying them, do not reveal any fault. But, Criterium C_2 and C_4 are *valid* because tests, satisfying them, reveals a fault. Therefore, tests t_i and t_j , satisfying criterium C_2 and C_4 are *reliable* and *valid*, thereby constitutes *ideal test* requirements.

V.A Demonstrating Example

This section discusses the demonstrating example implemented in Verilog HDL, and modeled by an FSM. As we can see in Fig. 3, four traffic signals and four lanes are present. Every lane has a separate traffic light with red, yellow and green lights. Verilog HDL implementation of TLC was realized by Xilinx Basys 3 Artix-7 FPGA development board and using Vivado 2017.4 design suite [17].

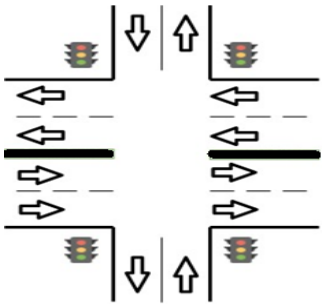

Fig. 3. Block Diagram of Traffic Light Controller

```
module Lights(n_lights,s_lights,e_lights,w_lights, clk_point1hz, btn, segment,input_ligth_status );

    output reg[2:0] n_lights,s_lights,e_lights,w_lights;
    input clk_point1hz;
    input btn;
    wire d1,d2;
    output reg [6:0] segment;
    integer state =0;
    input [11:0] input_ligth_status;
    integer output_ligth_status = 4444;

        always @(posedge clk_point1hz )
          begin

          if (btn ==0)
          begin
          state = 4'b0000;
          // input_ligth_status = 4442;
           output_ligth_status = 4444;
           n_lights = 3'b100;
           s_lights = 3'b100;
           e_lights = 3'b100;
           w_lights = 3'b100;
          end

            case (state)
            4'b0000:

                begin
                $display("The value of input: %d", input_ligth_status) ;
                segment = 7'b0000001;
                if (input_ligth_status == 12'b100100100010 )
                    begin
                    state=0;
                    output_ligth_status = 1444;
                    //input_ligth_status = output_ligth_status;
                    n_lights = 3'b001;
                    s_lights = 3'b100;
                    e_lights = 3'b100;
                    w_lights = 3'b100;
                    end
                else if ( input_ligth_status == 0)
```

Fig. 4. An excerpt code for TLC in Verilog HDL

An excerpt code of HDL for fault-free DUT is given in Fig. 4 which is implemented in Verilog. Original code is consisting 117 lines and nine states in always block.

A. Testing Preparation Step

In testing preparation step, the FSM model is extracted from HDL of TLC manually. The FSM modeling the behavior of TLC is converted into an RE, considering encoding of input/output combinations to the symbols given in TABLE II. The JFLAP tool [14] is used, see Fig. 5, to carry out the conversion of the FSM into corresponding RE. In Fig. 5, the FSM model of TLC, which is fault-free model, is given. The FSM is containing nine states and 18 transitions.

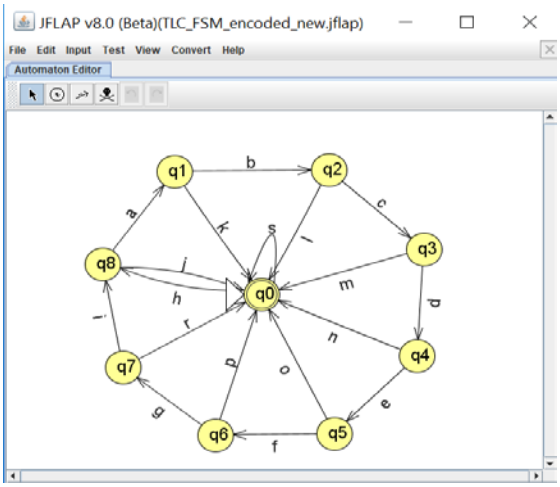

Fig. 5. FSM of TLC

TABLE II contains symbols, which are labels of the fault-free FSM model and encodings of input/output combinations. For example, the symbols "a" is encoding of "grrr 0 / yrrr" in which "grrr 0" is 13 bits input and "yrrr" is 12 bits output. In real circuit, "g", "y", and "r" represent "001", "010", and "100" in binary format respectively. "X" represents don't care condition for which "xxxxx - grrr 0" refers to set of 13 bits don't care different than "0011001001000".

TABLE II. ENCODING OF TRANSITIONS

Test Case Id	Test Input/ Expected Output	Test Case Id	Test Input/ Expected Output	Test Case Id	Test Input/ Expected Output
a	grrr 0 / yrrr	**g**	rrrg 0 / rrry	**n**	xxxxx - rrgr 0 / rrrr
b	yrrr 0 / rgrr	**i**	rrry 0 / grrr	**o**	xxxxx - rryr 0 / rrrr
c	rgrr 0 / ryrr	**j**	xxxxx - grrr 0 / rrrr	**p**	xxxxx - rrrg 0 / rrrr
d	ryrr 0 / rrgr	**k**	xxxxx - yrrr 0 / rrrr	**r**	xxxxx - rrry 0 / rrrr
e	rrgr 0 / rryr	**l**	xxxxx - rgrr 0 / rrrr	**s**	xxxx 1 / rrrr
f	rryr 0 / rrrg	**m**	xxxxx - ryrr 0 / rrrr	**h**	xxxx 0 / grrr

Test sequences are generated from RE in (4). However, only following test sequences t_1 and t_2, given in (5) and (6), are selected to use in test execution step.

R=[(h(abcdefgi)*(abcdefgr+abcdefp+abcdeo+abcdn+abcm+abl+ak+j)+s)*] (4)

t_1 = [h a b c d n] (5)

t_2 = [h a b c d e o] (6)

B. *Test Execution Step*

Exemplary, two faults are injected into FSM by combination of insertion/omission operators and related errors are injected into the HDL program to realize code-based mutation.

Mutant 1, depicted in Fig. 6, is representing the combination of missing state and transition error models. Therefore, state 3 and corresponding transitions are omitted to model this type of error.

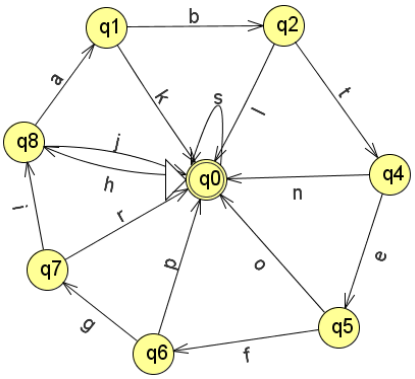

Fig. 6. FSM of mutant 1

Mutant 2, depicted in Fig. 7, is representing extra state and transition error models. State 9 and corresponding transitions are inserted to the original FSM. Extra transitions are u, v, and y which are encoded with “ryrr 0/rrgg”, “rrgg 0/ rrgr”, and “xxxxx – rrgg 0/ rrrr” respectively.

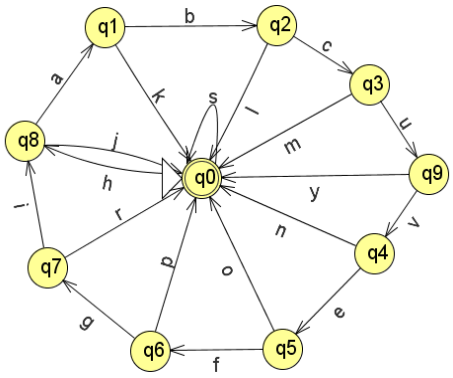

Fig. 7. FSM of mutant 2

Following, the corresponding FSMs are constructed and converted to REs. In this way, the exemplary test sequences, t_3 and t_4, are generated from the mutants.

t_3 = [h a b c t n] (7)

t_4 = [h a b c u y] (8)

Following results are collected from given positive and negative tests.

TABLE III. RESULTS FROM NEGATIVE AND POSITIVE TESTING

Test Sequences	**Fault-Free Device**	**Mutant Device 1**	**Mutant Device 2**
t_1	Test passed!	Test failed!	Test failed!
t_2	Test passed!	Test failed!	Test failed!
t_3	Test failed!	Test passed!	-
t_4	Test failed!	-	Test passed!

It is worth to note that the generated test sequences are executed on original and mutant Verilog implementation by simulation. Higher order mutants are generated from the behavioral HDL to model corresponding error types in FSM.

Test sequences t_1 and t_2 generated from the fault-free model are applied to the fault-free DUT and passes successfully but they fail when we apply them to the mutants. Test sequence t_3 and t_4 fail when we apply them to the fault-free DUT and pass when we apply them to the mutants. Therefore, test sequences t_1 and t_2 satisfy criterium C_2 when applying on mutants. Test sequences t_3 and t_4 satisfy criterium C_4 when applying on the fault-free DUT, see TABLE III. Therefore, these test sequences satisfy corresponding criteria and fulfill requirements of ideal testing defined in Section 4.

VI. Conclusion and Future Work

In this paper, *Ideal* testing of sequential circuits, represented by their HDL at behavioral level, is introduced. The proposed approach is based on mutation testing and formal models borrowed from Automata Theory. The first step, testing preparation, of the approach leads to a model of the given HDL represented by a FSM. This FSM is then converted to a RE. The same procedure is applied to the mutant models which are acquired by injecting faults into the original HDL model. In the second step, namely test executions, the resulting RE models are used. Tests are generated from fault-free and faulty (mutant) models that are used for positive and negative testing, respectively, to fulfill the requirements of ideal testing.

A traffic light controller example is used to demonstrate and validate the approach. Results of the experiment confirm that requirements of ideal testing are fulfilled.

Briefly, the contributions of the proposed approach are listed:

- Realizing an Ideal Testing framework for testing HDL behavioral model to target design errors
- Combining Ideal Testing concept with model-based and code-based Mutation testing techniques
- Demonstrating the proposed framework with the traffic light controller model.

The advantages of the proposed approach are:

- Ideal testing guaranties coverage of modeled faults in positive and negative test generation based on mutation testing
- Ideal testing proposes construction of reliable and valid tests in terms of defined requirements
- It offers higher-level test generation based on RE which provides higher abstraction, compactness and conformity compared to traditional test generation algorithms.

Despite testing at behavioral level which is a much higher level of abstraction than gate level, resulting much less components in our approach, still there is a limitation in the scalability. It is possible to encounter problem of state space explosion if the DUT becomes very large. To tackle this problem, in our future work, we intend to use model refinement [18] and/or model decomposition [19] to extend this and increase the scalability of the proposed methodology.

In the scope of this study, we addressed the errors modeled at the behavioral level, including extra state, missing state, extra transition, missing transition, and operation errors. As one of our future works, we plan to cover some of the faults at the gate level such as stuck-at faults. Thereby, we plan to use a fault simulator, and run the generated tests.

Finally, to evaluate the proposed approach, we aim at conducting experiments with other testing approaches such as random testing [20] and W-method [21] to make a comparison among the results.

REFERENCES

[1] A. Valmari, "The state explosion problem." Lectures on Petri nets I: Basic models. Springer Berlin Heidelberg, 1998. 429-528.

[2] S.C. Kleene, Representation of events in nerve nets and finite automata. No. RAND-RM-704. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[3] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. Eric Wong. "Model-based mutation testing—approach and case studies." Science of Computer Programming 120 (2016): 25-48.

[4] J.B. Goodenough, and S.L. Gerhart. "Toward a theory of test data selection." IEEE Transactions on software Engineering 2 (1975): 156-173.

[5] W.E. Howden, "Reliability of the path analysis testing strategy." IEEE Transactions on Software Engineering 3 (1976): 208-215.

[6] L. Bougé, "A contribution to the theory of program testing." Theoretical Computer Science 37 (1985): 151-181.

[7] M. Bushnell, and V. Agrawal, Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits. vol. 17. Springer Science & Business Media (2004).

[8] T. Bengtsson, and S. Kumar, A survey of high level test generation: Methodologies and fault models. Ingenjörshögskolan, Högskolan Jönköping (2004).

[9] G. Jervan, Z. Peng, O. Goloubeva, M.S. Reorda, and M. Violante, "High-level and hierarchical test sequence generation" ,In: Seventh IEEE International High-Level Design Validation and Test Workshop, pp. 169-174 (2002).

[10] M. Lajolo, M. Rebaudengo, M.S. Reorda, M. Violante, and L. Lavagno, "Behavioral-level test vector generation for system-on-chip designs", In: Proceedings IEEE International High-Level Design Validation and Test Workshop, pp. 21-26 (2000).

[11] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral VHDL models", Proceedings International IEEE Test Conference (1998).

[12] T.B. Nguyen, and C. Robach, "Mutation testing applied to hardware: mutants generation", In Proceedings of the 11th IFIP International Conference on Very Large Scale Integration, pp. 118-123(2001).

[13] C. Robach, and M. Scholive, Simulation-Based Fault Injection and Testing Unsing the Mutation Technique. Fault injection techniques and tools for embedded systems reliability evaluation. Springer, Boston, MA, 195-215 (2003).

[14] F. Belli, "Finite state testing and analysis of graphical user interfaces", Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on. IEEE, (2001).

[15] F. Belli, M. Beyazit, and N. Güler, "Event-Oriented, Model-Based GUI Testing and Reliability Assessment—Approach and Case Study", Advances in Computers, 85, 277-326, (2012).

[16] S. Rodger, and T. Finley, JFLAP - An Interactive Formal Languages and Automata Package, ISBN 0763738344, Jones and Bartlett (2006).

[17] Xilinx Vivado, Available online: https://www.xilinx.com/products/design-tools/vivado.html (last access: Jan 2018).

[18] F. Belli, N. Güler, and M. Linschulte, "Layer-centric testing", FERS-Mitteilungen: vol. 30, no. 1 (2012).

[19] S. Devadze, E. Fomina, M. Kruus, and A. Sudnitson, "Web-based system for sequential machines decomposition", In: EUROCON 2003, The IEEE Region 8, vol. 1, pp. 57-61 (2003).

[20] H. Richard, "Random testing", Encyclopedia of software Engineering (1994).

[21] T.S. Chow, "Testing software design modelled by finite state machines", IEEE Transactions on Software Engineering, vol. 4, no. 3, pp. 178-187 (1978).