Regular Expression Based Test Sequence Generation for HDL Program Validation

Onur Kilincceker
University of Paderborn, Paderborn, Germany.
Mugla Sitki Kocman University, Mugla, Turkey.
okilinc@mail.upb.de

Ercument Turk
International Computer Institute, Ege University, Izmir, Turkey.
ercument_turk@hotmail.com

Moharram Challenger
International Computer Institute, Ege University, Izmir, Turkey.
moharram.challenger@ege.edu.tr

Fevzi Belli
University of Paderborn, Paderborn, Germany.
Izmir Institute of Technology, Izmir, Turkey.
belli@upb.de

***Abstract*— This paper proposes a test sequence generation approach for behavioral model validation of sequential circuits implemented in Hardware Description Language (HDL). In the procedure of test sequence generation proposed in this study, Regular Expressions (REs) are utilized to model the behavior of the System Under Test (SUT). First, the HDL program is converted to a Finite State Machine (FSM). Then, the obtained FSM is transformed to RE which is represented by a Syntax Tree (ST). In this way, the test sequence generation problem is simplified to the tree traversal algorithm in which symbol and operator coverage criteria are satisfied. The required tools for test sequence generation are provided to automatize the whole procedure of the proposed approach. Also, a running example, based on a real-life-like Traffic Light Controller (TLC), validates the proposed approach and analyzes its characteristic features.**

Keywords— regular expression; test sequence generation; hardware design validation; behavioral model; hardware description language

I. INTRODUCTION

According to the well-known Moore's Law, the number of transistors or components on a Very Large-Scale Integration (VLSI) chip doubles approximately in every 18 months. Therefore, the testing and validation of hardware become increasingly critical and overwhelming. As a result, the emergence of new methods to handle this problem in an efficient way becomes a necessity.

The validation of hardware in early stages of the design flow not only reduces the cost, due to low complexity considering the abstraction level, but also provides reusability of test sequences at lower design levels. Generally, test sequences or patterns are generated for two purposes: validation testing to target design faults; and structural testing to address manufacturing faults. The first one can be applied at the early stage of the design flow, i.e., behavioral design, and the generated test sequences can be reused at lower levels, e.g., register transfer level or gate level. The second one, structural testing, is applied at the gate level and requires extensive analysis. Thus, it demands a high cost for test pattern generation. On the other hand, the structural testing provides appropriate and effective coverage of targeted manufacturing faults that are commonly stuck-at 0/1. In these faults, signals or pins are assumed to be stuck at logical '1' or '0'. It is reported in [1] that there is a direct correlation between design faults in the behavioral level and manufacturing faults in the gate level as illustrated in Figure 1. Therefore, the test sequence generation approach discussed in this study can be useful for targeting design faults as well as manufacturing faults.

This paper proposes a methodology for test sequence generation to validate a given HDL program (as a Finite State Machine (FSM)) and target design faults at the behavioral level. The FSM model is automatically extracted from the HDL program by scanning the code to find the state and transition patterns. Then, this FSM is converted into a RE, which offers more abstraction, compactness and conformity to algebraic operations based on Kleene Algebra [2]. This RE is represented by a Syntax Tree (ST). Operators and symbols of RE are represented in the external and internal nodes of the ST, respectively. As a result, the procedure of the test sequence generation is carried on by traversing this ST. The proposed approach is evaluated on a real-life-like traffic light controller example.

Figure 1. Relation of Gate and Behavioral Level Faults [1]

Next section presents related work covering studies on validation testing at behavioral and RTL levels and test sequence generation based on RE. Section 3 introduces the background including used notions and coverage criteria. Section 4 explains the proposed approach which is demonstrated by an example, a traffic light controller, in Section 5. In Section 6, the results are presented and discussed. Finally, Section 6 concludes the paper and gives the highlights of our further research.

II. RELATED WORK

Conventional validation approaches for hardware design utilize combination of simulation-based techniques and formal verification methods [3]. However, coverage oriented/driven test generation, which is extensively used in software testing, is becoming popular for hardware design validation at high-level of abstraction. HYBRO [4] works at Register Transfer Level (RTL) based on branch-coverage directed approach. It uses Satisfiability Modulo Theory (SMT) solver and requires dynamic and static analysis of RTL source code for generation of test

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00103

sequences. Therefore, the computation cost of SMT solver and the source code analysis limits the effective generation of test sequences in desired time and may results in long test sequences.

Branch-oriented Evolutionary Ant Colony OptimizatioN (BEACON) [5] also works at RTL and uses an algorithm to increase search capacity of design space. It is also based on branch coverage and converts HDL program to C++ code for fast simulation. However, lack of proper guidance of simulation results in long test sequences and requires high time consumption [6]. Particle Swarm Optimization-based Functional Test (PSOFT) generator [6] utilizes an algorithm to reach corner cases of design and uses branch coverage for design validation at RTL. It is combined with a controlled graphical search method. Thus, PSOFT is highly resource consuming for generation of effective test sequences in limited time. On the other hand, RTL level test sequences may result in high coverage of manufacturing faults at the gate (structural) level. Benefit from abstraction offers diminishing cost depending on decreasing complexity of HDL program. Therefore, the current work uses behavioral level instead of RTL or gate level for this reason.

Test sequence generation algorithms at behavioral level are varying depending on specific coverage metrics. For example, bit and condition coverages are proposed in [9]. Bit coverage assumes that each bit of variable or signal in the design may be stuck at 0/1. Condition coverage assumes that each condition in the design may be stuck at True/False. In [9], a high correlation between bit coverage at the behavioral level and stuck-at fault coverage at the gate level is found whereas it is reported in [1] that test sequence generation at the behavioral level reduces the costs and increases the final product quality.

Jervan at al. [7] analyze existing coverage metrics at the behavioral level with correlation of gate level stuck-at faults. Results from the analysis show that the combination of bit and condition coverage can be effectively utilized for evaluating the quality of a test set at the behavioral level. Moreover, Jervan at al. [7] use Random Mutation Hill Climber (RMHC) algorithm for test sequence generation. The RMHC is based on random generation of test sequences starting from given initial solution whose neighbors are examined to find the next solution. Procedure continues until the pre-defined iteration is completed. As another study, Kilincceker et al. [8] elaborate an approach to apply the ideal testing on HDL programs at the behavioral level. To this end, they provide a testing methodology in conjunction with mutation testing. They also combine positive and negative testing to fulfill requirements of the ideal testing with novel test selection criteria.

Regular expression provides higher abstraction, compactness and conformity to algebraic operations based on Kleene Algebra [1]. Brzozowski [10] uses regular expression for modeling sequential circuits. He provides an algebraic algorithm for construction of a regular expression from given finite state machine. In addition, in his seminal work [11], he introduces derivatives of regular expression.

In recent years, regular expression became more popular for software testing. Belli and Grosspietsch [12] propose integration of fault tolerance properties into the design of complex software systems by utilization of regular expression. They also define insertion, deletion, and replace operators to model faulty behavior of the software system. These operators can be also used in the generation of mutations of given regular expression. Arcaini et al. [13] provide a fault-based algorithm for test sequence generation, which uses regular expression in conjunction with mutation testing. They introduce several mutation operators and implement proposed algorithm in a tool called MuTreX [14] to generate fault-detecting sequences.

Mariani et al. [15] propose alphabet, operator, and expression coverage criteria for self-testing of software components using RE. However, they do not provide any test generation methodology using defined coverage criteria. Current work uses the criteria in [15] for generation of test sequences. Liu and Miao [16] offer theory of test modeling based on regular expression for software behavior. Liu et al. [17] introduce an extended regular expression for modelling and testing software behavior. They define some rules for modeling given software using the extended regular expression to generate test sequences from this novel model and detect program errors.

Ravi et al. [18] utilize regular expression for testability analysis and optimization at RTL controller/data path circuits. In addition to the conventional use of regular expressions, they represent symbols of Boolean functions instead of input/output combinations and generate test sequences for predefined actions. However, the authors indicate that their methodology unifies testability analysis for several types of digital circuits. Current work uses regular expression for test sequence generation to validate the given HDL program at the behavioral level. This test sequence generation approach conforms to some coverage criteria to target design faults, including bit stuck-at 0/1.

III. BACKGROUND

In this section the terminologies which are used in this paper as well as the coverage criteria are discussed. The notations and models used in the current work, including Finite State Machine (FSM), Regular Expression (RE), Extended Regular Expression (ERE), and Syntax Tree (ST), are elaborated in the first subsection. In the following subsection, the coverage criteria for the defined models are presented.

A. Used Notions

Finite State Machine **(FSM):** A FSM is defined by 5-tuples [19] as $<Q, \Sigma, \delta, q_0, F>$ where,

- Q: a finite set of states
- Σ: a finite set of input symbols (alphabet)
- δ: state transition function, mostly represented by a table
- q_0: an initial (starting) state which is an element of Q
- F: a finite set of final states which is a subset of Q.

Regular Expression **(RE)** [20]: RE is basically a sequence of symbols connected by following operators.

- *Concatenation* – is represented by '.' or without any specific symbol. For example, "*a.b"* or "*ab"* means that symbol a is followed by symbol b.
- *Selection (Union)* - is represented by '+' operator. For example, *a+b* means '*a or b*'.
- *Iteration (Kleene's Star)* - is represented by '*' operator. For example, "*a**" means *'a is iterated infinite times'* (including zero times which represents *empty word* "*λ"*). Similarly, a^+ means at least one-time iteration, that is excluding *λ*.

Example 1: An example regular expression is given below;

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

This regular expression means that a symbol "*a"* is followed by a symbol "*b"* that is followed by zero or more times iteration of symbol "*c"* or "*d"*.

Extended Regular Expression (ERE) [17]: The definition of regular expression given above is extended using the following range operator applied to Kleene's Star operator.

- *Range* - is represented by "n~m" instead of "*" operator. For example, "$a^{1\sim2}$" means that one to two times iteration included.

Example 2: The regular expression given in example 1 can be extended as follows:

$$[(a\ \ b\,(c+d)^{1\sim2})] \quad (2)$$

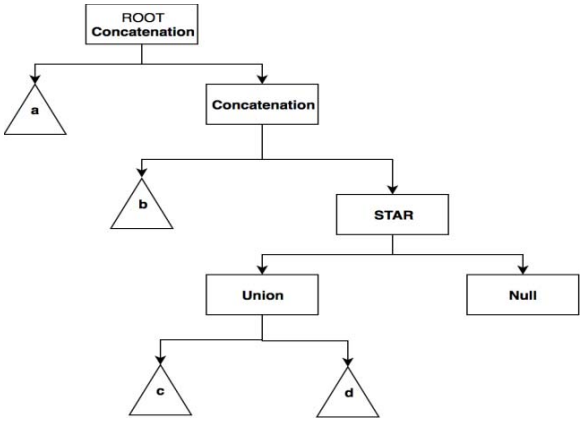

Figure 2. The ST of the RE given in Example 1

Syntax Tree (ST): A syntax tree is basically a tree to represent a regular expression. It can be either right-to-left or left-to-right associative. Root and internal nodes store operators and leaves store symbols of the regular expression. The ST representation, left-to-right associative, of RE in Example 1 is shown in Figure 2.

The set of test sequences generated from the ST model, using the approach proposed in this study, constitutes a test suite. Depending on the selection of consecutive Kleene's star operator, the length of any test sequence can be infinite. To tackle this problem, there is a solution requiring utilization of the range operator based upon the ERE model. The range operator can be defined using any specific values.

B. Coverage Criteria

There are several coverage criteria to assess adequacy of the test. They define how the system is tested thoroughly and whether the generated test sequences are good enough. In software testing perspective, "thorough", "good enough", and "adequate" refer to same meaning [21]. Coverage criteria are used to achieve these quality measurements. A test set is adequate when it satisfies particular coverage criteria.

The bit coverage defined for HDL program validation at the behavioral level is used in this study.

Bit coverage [9]: It is satisfied if for each bit of a variable, signal, and port in the behavioral design, there is at least a test sequence which exercises bit stuck-at 0/1 assignments of these bits.

The alphabet and operator coverage criteria are defined for test sequence generation based on RE in the literature [14].

Alphabet coverage [15]: It is satisfied if for each symbol 'a' in the alphabet, there is at least a test sequence, including symbol 'a', in the test suite. For instance, the test suite {abcd} satisfies this criterion for the RE given in Example 1.

Operator coverage [15]: It is satisfied if for each union operator, the test suite includes a test case containing the first operand and another test case containing the second operand of the union operator.

For each Kleene's star operator, the test suite includes a test sequence containing no iteration, exactly one iteration and more than one iteration of operand of the Kleen's star operator. For instance, the test suite {*ab, abc, abd, abcc, abcd, abdd*} satisfies this criterion for the RE given in Example 1.

IV. Approach

The proposed approach includes the following steps: extraction of FSM from the given HDL program, conversion of FSM to RE, ST construction from RE, traversal of ST to generate test sequences satisfying the alphabet and operator coverage criteria. Figure 3 depicts these steps.

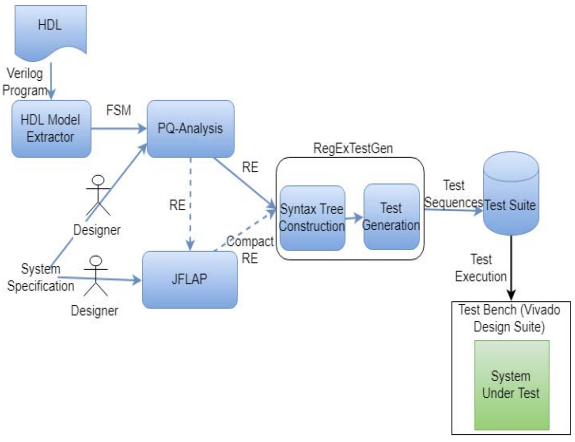

Figure 3. General overview of the proposed approach

In the FSM extraction step, an HDL program is analyzed to generate a FSM. Note that only HDL programs, which satisfy specific patterns, can be taken into consideration. The FSM Extraction program reads the HDL program to determine states and transitions.

In the FSM to RE conversion step, the well-known Brzozowski [10] algorithm is used, which requires to

construct an equation system from the state transition table of FSM. This equation system defines transition relations of states. The system is represented by a square matrix on which an elimination algorithm, e.g. Gaussian elimination algorithm, is run to generate the RE. Arden's rule [22] is applied on these equation system to eliminate redundant transitions and shorten the system. The application of this elimination continues until obtaining one column and one row for equation which represents the expected RE.

Based on the proposed approach, the FSM to RE conversion can be done using PQ-Analysis tool [23]. However, this conversion may result in a longer RE. Therefore, the JFLAP tool [24] can be used to shorten this RE as depicted in Figure 3. This approach provides a compacted RE model to reduce the computation in the further steps.

Note that in some systems, the HDL program is not already available. So, the model extraction phase of the proposed approach is not useful in these cases. However, the hardware specification of the system, such as state machine diagram, activity diagram, and sequence diagram, may be available and provide the behavior model for the system. As these specifications imply dynamic of the system, the designer can create the FSM for the system using tools such as PQ-Analysis [23] or JFLAP [24].

As a result, the proposed approach supports both creating the FSM from the system specification and extracting it from the given HDL program.

For the ST construction, a tool called dk.brics.automaton [25] is used. This tool provides an open source implementation of an antichain algorithm called forward subset. In our study, this tool has been extended in a way that it gets rid of infinity Kleene's Star operator.

Based on the proposed approach, the test sequence generation from RE requires traversing the ST. For example, the following test sequences can be generated from the given RE in (1) by traversing the corresponding ST.

$$\{\ ab,\ abc,\ abd,\ abcc,\ abdd,\ abcd,\ \ldots\ \} \quad (3)$$

According to the Kleene's Star operator, the length of the test sequences can be infinite (see (3)). This infinity can be resolved by utilization of the ERE model. For example, the ERE given in (2) can be used for (1). In this example, the iteration of the Kleene's star is bounded to 1-2 by utilizing a range operator. Therefore, the following test sequences can be generated;

$$\{ab,\ abc,\ abd,\ abcc,\ abdd,\ abcd\} \quad (4)$$

The following algorithm defines the procedure of test sequences generation in this study. The input of the algorithm is the ST of the RE model. The output is the resulting set of test sequences by traversing the ST.

This algorithm starts with the root node of the ST and traverses the left and right nodes based on the given procedure. The following values can be encountered in each node during traversal of the ST;

- A Symbol: The symbol is added to the current test sequence because it is a leaf node.
- A Union operator: The test sequences from the left and right nodes of union operator are combined and returned as the result.
- A Star operator: As it is already discussed, 0 and 1 iterations are considered for star operator to satisfy the operator and alphabet coverage criteria. Therefore, test sequences including empty word and the string set from the left node of the current node are combined as the result of this operator.
- A Plus operator: The same procedure of the star operator is applied for the plus operator excluding empty word, as it only contains 1 iteration.
- A Concatenation operator: This operator concatenates each string of the left node with each string of the right node.
- A Range (n~m) operator: In this operator, the lower bound of the rage is considered to cover the criteria and extra iterations are avoided. Therefore, concatenation of the operand for n times is the result of this operator.

Algorithm: Test sequence generation based on ST of RE

Input: S // S is syntax tree of given RE
Output: ti ϵ T, i=1,…,n//T is the resulting test suite

```
node = ST.root
Set RegExTestGeneration (SyntaxTree node) {
    if (node.data == symbol)
        return {node.data}
    if (node.data == union)
        return RegExTestGeneration(node.left) U
                RegExTestGeneration (node.right)
    if (node.data == Star)
        return RegExTestGeneration (node.left) U Epsilon
    if (node.data == Plus)
        return RegExTestGeneration (node.left)
    if (node.data == Concatenation)
        return Concatenation(
                    RegExTestGeneration(node.left),
                RegExTestGeneration (node.right))
    if (node.data == 'n~m'){
        Set s = RegExTestGeneration (node.left)
        for (i=0 to n)
                Set p = Concatenation (p, s)
                return p
    }
}
Set Concatenation (Set s1, Set s2){
    Set result = empty
    for   each item1 in s1
        for each item2 in s2
                result = result U item1.item2
}
```

After the execution of one of the above-mentioned conditions for each node, based on the node's data, the traversal continues with the predecessor operator node using the test sequence set already generated.

A tool called RegExTestGen, demonstrated in Section V, is developed by implementing the proposed test sequence generation algorithm. The RegExTestGen takes the RE and construct the ST from which test sequences are generated.

V. A Running Example

This section discusses a running example called Traffic Light Controller (TLC). To this end, the System Under Test (SUT) in this study is TLC which is implemented in Verilog HDL. As it can be seen in Figure 4, four traffic signals and four lanes present a real-life-like traffic light. 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 [26].

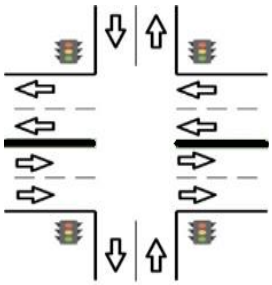

Figure 4. Block Diagram of Traffic Light Controller (TLC)

An excerpt code of the HDL for the TLC is given in Figure 5. The TLC is implemented in Verilog HDL and the source code contains 117 lines and nine states in an always block using case statement.

```
`timescale 1ns / 1ps
...
module Lights (n_lights,s_lights,
e_lights,w_lights, clk_point1hz, btn,
segment,input_ligth_status,
output_ligth_status);
...
always@(clk_point1hz)
begin
case (state)
 4'b0000:
 begin
   $display("The value of input:%h,
    state%d",input_ligth_status,state) ;
   segment = 7'b0000001;
   if (btn==0)
   begin
     state = 4'b1000;
     output_ligth_status = 16'h1444;
     n_lights = 3'b001;
     s_lights = 3'b100;
     e_lights = 3'b100;
     w_lights = 3'b100;
   end
   else if (btn==1)
   begin
     state = 4'b0000;
     output_ligth_status =
       16'h0000;
     n_lights = 3'b100;
     s_lights = 3'b100;
     e_lights = 3'b100;
     w_lights = 3'b100;
   end
   else
   begin
     state = 4'b0000;
     output_ligth_status =
       16'h4444;
     n_lights = 3'b100;
     s_lights = 3'b100;
     e_lights = 3'b100;
     w_lights = 3'b100;
   end
end
4'b0001:
  …
4'b0010:
  ...
4'b0011:
  ...
4'b0100:
  ...
4'b0101:
  ...
4'b0110:
  ...
4'b0111:
  ...
4'b1000:
  ...
endcase // case (state)
end
endmodule
```

Figure 5. An excerpt code for TLC in Verilog HDL

To apply the proposed approach, in the first step, the FSM model is extracted from HDL of TLC automatically. The extraction tool scans each line of the Verilog code to find particular pattern. These patterns are checked to figure out whether they satisfy pre-defined rules or not. The lines satisfying the rules are used to define states of the FSM. The transitions of the FSM are also extracted from the relations between states in the HDL program.

TABLE I. Encoding of transitions. Key: g=Green; y=Yellow; r=Red

Symbol	I/O Combination	Symbol	I/O Combination	Symbol	I/O Combination
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

The extracted FSM, modeling the behavior of TLC, is converted into a RE, considering encoding of input/output combinations to the symbols given in TABLE I. The FSM of original HDL contains 9 states and 18 transitions see Figure 6.

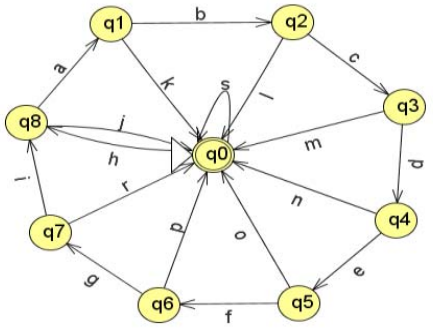

Figure 6. The FSM of TLC

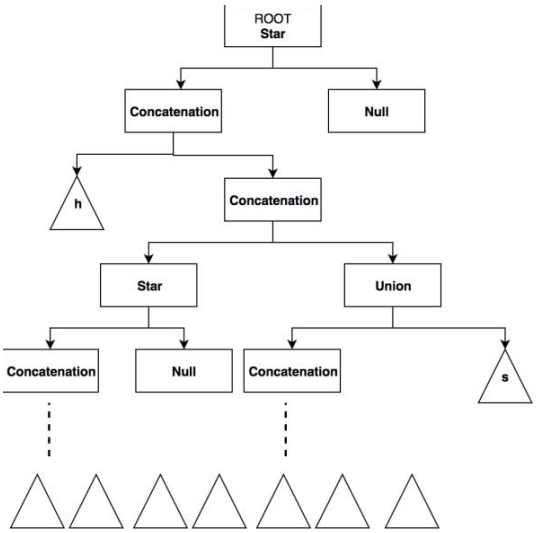

Figure 7. The ST of the RE in (5)

The extracted FSM model is directly imported to PQ-Analysis [23] tool for the RE conversion. Resulting RE is compacted by JFLAP tool [24] and the RE (5) is obtained.

[*(h(abcdefgi)*(abcdefgr+abcdefp+abcdeo+abcdn+abcm+abl+ak+j)+s)*](5)

The representation of the RE, given in (5), is carried out by utilization of the ST in which external nodes, i.e. leaves, store symbols whereas internal nodes along with root store operators. The ST of the RE in (5) is shown in Figure 7 where root node is a star operator. The number of external nodes equals to number of symbols. Each symbol contained in the RE is stored in an external node of the ST. The height of the tree roughly equals the number of operators in the worst-case scenario.

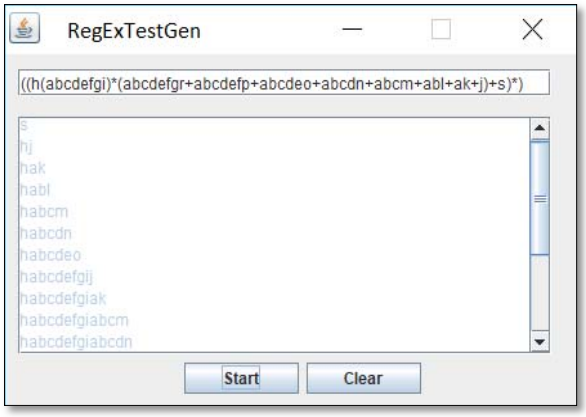

Figure 8. A Screenshot of RegExTestGen Tool

Test sequences are generated from the ST by traversal algorithm given in Section 4. For the ST construction and the test sequence generation, RegExTestGen tool, see Figure 8, is used. The details of the algorithm used for test sequence generation are given in Section IV.

The RegTextGen tool results in test sequences satisfying alphabet and operator coverage criteria. The generated test suite, which is a set of test sequences, is given in TABLE II. Total length of the test suite is 162.

TABLE II. GENERATED TEST SEQUENCES

Test Sequences			
t_1	habcdefgiabcdefgr	t_{11}	habcdefp
t_2	habcdegiabcdep	t_{12}	habcdeo
t_3	habcdefgiabcdo	t_{13}	habcdn
t_4	habcdefgiabcdn	t_{14}	habcm
t_5	habcdefgiabcm	t_{15}	habl
t_6	habcdefgiabl	t_{16}	hak
t_7	habcdefgiak	t_{17}	hj
t_8	habcdefgij	t_{18}	s
t_9	habcdefgr		
t_{10}	habcdefp		

The test sequences are executed on the TLC via test bench using simulation environment provided by Vivado 2017.4 [26]. The test bench software written in HDL reads the test entries from a file and applies them to the TLC system and then analyzes the system's outputs. The details of collected results from the test execution are given in Section VI.

VI. RESULTS AND DISCUSSION

To evaluate effectiveness of proposed approach, a tool called GraphWalker [27] is used for random test sequence generation with the edge and vertex coverage criteria. Also, a bit stuck-at fault model is used to measure fault coverage of generated test sequences of both proposed approach and random test sequence generation. Therefore, the output bit stuck-at 0/1 faults and case bit stuck-at 0/1 faults are simulated for test sequence execution. Statistics of simulated fault models is given in TABLE III.

TABLE III. FAULT STATISTICS

Fault Types		Number of Design Faults
Bit Stuck-at	Output Bit Stuck-at 0	12
	Output Bit Stuck-at 1	12
	Case Bit Stuck-at 0	3
	Case Bit Stuck-at 1	3
Total		30

To do an evaluation, a comparison between the proposed approach, with alphabet and operator coverage criteria, and random testing method, with edge and vertex coverage criteria, is realized. Considering the proposed approach, the test sequences are generated for the TLC case study using RegExTestGen tool. For random testing, test sequences were generated using GraphWalker. To have the same condition for both approaches, the tools are adjusted to have %100 coverage for related criteria to assess adequacy of the test. Therefore, the edge and vertex coverage criteria of random testing was set to %100. To eliminate the effect of randomness on test sequence generation time, the GrapWalker was run 10 times and the average value is taken. The test suits generated by the two approaches are executed automatically using the provided test bench in Vivado design suite.

The results show that the current approach is more efficient than the random method in terms of both length of the test sequences and the required time for test sequence generation, see TABLE IV. The fault coverages of the two approaches are %100 for 30 faults. However, the length of the test suite for random testing is bigger than length of the test suite for the proposed approach. This leads to increase of test generation and execution time for random testing comparing to the proposed approach.

With maximum edge coverage for random testing the generated test suite is about 3 times longer than the one generated by proposed approach. This is about 42 times

longer when the maximum vertex coverage is used for random testing, see Figure 9 for the comparison.

TABLE IV. COMPARISON OF RESULTS

	Graph Walker		Proposed Approach
	Edge Coverage	Vertex Coverage	Operator and Symbol Coverage
	%100	%100	%100
Fault Coverage (%)	**100**	**100**	**100**
Length of Test Sequences (Symbols)	532	6816	**162**
Time for Test Generation (ms)	4789	4045	**611**
Time for Test Execution (ns)	18785	55320	**2370**

As you can see in Figure 9, the proposed approach has about 8 times faster test execution time than the random testing with full edge coverage. The proposed approach is about 23 times faster than random testing when the maximum vertex coverage is used.

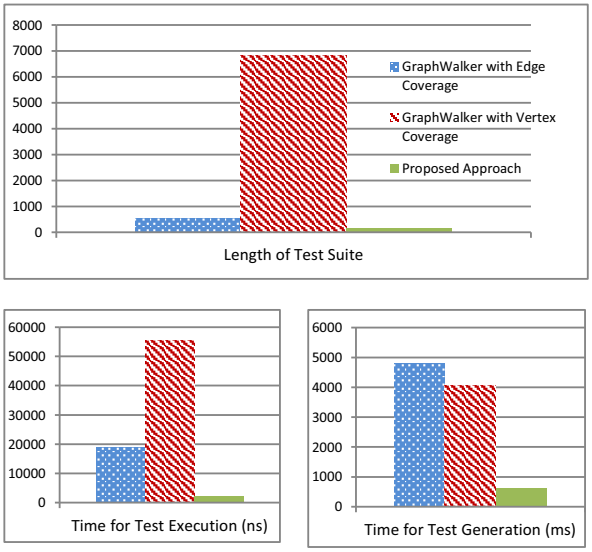

Figure 9: Comparison for the GraphWalker and the proposed approach

Please note that the random testing method used for the evaluation in this study is not a traditional random testing. Because, the current random testing method is more systematic with respect to the edge and vertex coverage criteria. This explains how to reach %100 fault coverage by means of the current random testing. Also, it is clear that the fault coverabe will be decreased if the traditional random testing method is utilized due to the generation of redundant test sequences which do not confrom to the system behavior.

VII. CONCLUSION AND FUTURE WORK

In the scope of this paper, an approach for test sequence generation using RE is introduced to target design faults at behavioral level. The proposed approach starts with extraction of FSM model from given HDL program. Then the obtained FSM is converted into RE by means of well-known Brzozowski algorithm [10]. Afterwards, this RE model is represented by ST from which the test sequences are generated using tree traversal algorithm considering pre-defined coverage criteria on RE.

A traffic light controller example is used to demonstrate and validate the approach. The resulting test sequences are executed on the SUT by means of a test bench provided by Vivado design suite [26]. The results of the experiment confirm that the proposed approach is efficient for the real-life-like example considering length of test sequences, time for test generation, and time for test execution.

Briefly, the contributions of the proposed approach can be listed as follows:

- Test sequence generation based on regular expression in terms of operator and alphabet coverage criteria to target design faults at behavioral level for validation testing
- Developing a tool chain to support entire methodology
- Supporting both available HDL programs and specification of system under development in the proposed approach

Within this study, we address the behavioral fault models including bit stuck-at 0/1. As one of our future works, we plan to cover some of the faults in the gate level such as stuck-at 0/1 faults. To do this, we plan to use a fault simulator in the gate level and apply generated test sequences from our approach to find out the fault coverage at the gate level.

Another future work is to decrease and adjust the alphabet and operator coverage for analyzing its impact on fault coverage. In this way, the suite can be tightened; thus, the cost of test generation and execution may reduce.

Finally, the proposed approach can be adapted with hierarchical modularization of HDL programs, which improves the scalability.

REFERENCES

[1] M. Lajolo, L. Lavagno, M. Rebaudengo, Sonza M. Reorda, M. Violante, "Behavioral level Test Vector Generation for System-an-Chip Designs", Proc. High level Design Validation and Test Workshop, pp. 21-26, 2000.

[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] Chen, Mingsong, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. System-level validation: high-level modeling and directed test generation techniques. Springer Science & Business Media, 2012.

[4] L. Lingyi, and S. Vasudevan,"Efficient validation input generation in RTL by hybridized source code analysis." In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp. 1-6. IEEE, 2011, doi: 10.1109/DATE.2011.5763253

[5] L. Min, K. Gent, and M. S. Hsiao,"Design validation of rtl circuits using evolutionary swarm intelligence." In Test Conference (ITC), 2012 IEEE International, pp. 1-8. IEEE, 2012, doi: 10.1109/TEST.2012.6401556

[6] P. Prateek, and M.S. Hsiao. "Fast stimuli generation for design validation of rtl circuits using binary particle swarm optimization." In VLSI (ISVLSI), 2015 IEEE Computer

Society Annual Symposium on, pp. 573-578. IEEE, 2015, doi: 10.1109/ISVLSI.2015.26

[7] 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, doi: 10.1109/HLDVT.2002.1224448

[8] O. Kilincceker, E. Turk, M. Challenger, and F. Belli, “Applying the Ideal Testing Framework to HDL Programs”, ARCS 2018 – 31st International Conference on Architecture of Computing Systems, 14th Workshop on Dependability and Fault Tolerance (VERFE'18), in press.

[9] F. Ferrandi, G. Ferrara, D. Scuito, A. Fin, F. Fummi, "Functional Test Generation for Behaviorally Sequential Models", Proc. Design Automation and Test in Europe, pp. 403-410, 2001, doi: 10.1109/DATE.2001.915056

[10] J.A. Brzozowski,"Regular expressions from sequential circuits." IEEE Transactions on Electronic Computers 6 ,pp. 741-744,1964.

[11] J.A. Brzozowski,"Derivatives of regular expressions." Journal of the ACM (JACM) 11.4, pp. 481-494,1964, doi: 10.1109/PGEC.1964.263932

[12] F. Belli, and K-E. Grosspietsch. "Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study." IEEE Transactions on software engineering 17, no. 6, pp.513-526,1991, doi: 10.1109/32.87278

[13] P. Arcaini, A. Gargantini, and E. Riccobene. "Fault-based test generation for regular expressions by mutation." Software Testing, Verification and Reliability, 2018.

[14] Code is available at https://github.com/fmselab/mutrex/ (last access: April 2018).

[15] L. Mariani, M. Pezze, D. Willmor,”Generation of integration tests for self-testing components”, In International Conference on Formal Techniques for Networked and Distributed Systems (pp. 337-350). Springer, Berlin, Heidelberg,2004, doi: 10.1007/978-3-540-30233-9_25

[16] P. Liu, and H. Miao. "Theory of test modeling based on regular expressions." In International Workshop on Structured Object-Oriented Formal Language and Method, pp. 17-31. Springer, Cham, 2013.

[17] P. Liu, J. Ai, and Z.J. Xu. "A study for extended regular expression-based testing." In Computer and Information Science (ICIS), 2017 IEEE/ACIS 16th International Conference on, pp. 821-826. IEEE, 2017, doi: 10.1109/ICIS.2017.7960106

[18] S. Ravi, G. Lakshminarayana, and N.K. Jha. "TAO: Regular expression-based register-transfer level testability analysis and optimization." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9, no. 6 (2001): 824-832, doi: 10.1109/TEST.1998.743171

[19] J.E. Hopcroft, R. Motwani, and J.D. Ullman. Introduction to automata theory, languages, and computation. Harlow: Pearson,2014.

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

[21] A.P. Mathur. Foundations of software testing, 2/e. Pearson Education India, 2013.

[22] D.N. Arden,"Delayed-logic and finite-state machines." Switching Circuit Theory and Logical Design, SWCT 1961. Proceedings of the Second Annual Symposium on. IEEE, 1961, doi: 10.1109/FOCS.1961.13

[23] PQ-Analysis Tool, available online: http://download.ivknet.de/ (last access: April 2018).

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

[25] Code is available at http://www.brics.dk/automaton/ (last access: April 2018).

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

[27] GraphWalker Tool, available online: http://graphwalker.github.io/ (last access: April 2018)