AAG: An Automatic Assertion Generation Framework for RTL Designs

Shahid Ali Murtza, Osman Hasan and Kashif Saghar
School of Electrical Engineering and Computer Science
National University of Sciences and Technology (NUST)
Islamabad, Pakistan
Email:{smurtza.msee15seecs,osman.hasan,kashif.saghar}@seecs.nust.edu.pk

***Abstract*—Assertion Based Verification (ABV) has been shown to be a very effective functional verification approach for digital designs. ABV is usually employed by the verification engineers by embedding assertions in the hardware description language (HDL) code manually by studying the design and user provided specifications. However, with the growing complexity of digital systems, understanding different designs and specifications in general and then writing assertions manually in particular has become quite tedious. In this paper, we propose to alleviate these issues by proposing AAG, i.e., an Automatic Assertion Generation framework that accepts the Register Transfer Level (RTL) code in Verilog, generates the corresponding randomized testbench automatically and then generates the corresponding value change dump (VCD) file from the simulation of RTL code using the generated testbench. In the proposed verification framework, we use GoldMine as an assertion generation engine. The paper also explains, with help of case studies, how can verification engineers benefit from AAG.**

I. INTRODUCTION

The functional verification of modern day hardware designs has become a great challenge due to their increasing design complexity, growing design size, and short time-to-market. Traditionally, the designers and verification engineers simulate designs written in hardware description languages (HDLs), like Verilog, along with their testbenches for functional verification. This approach is quite time consuming and usually results in missing the corner cases, since test benches are usually biased based on the inclinations of the test engineers. Moreover, the simulation approach cannot guarantee the absence of bugs in the design due to its inherent incompleteness. As a result, bugs are detected at a later stage of the design and verification flow, which can in turn lead to unforeseen scenarios, like the Intel Pentium bug [1].

Formal verification methods [2] have been advocated to overcome the above-mentioned limitations of simulation based verification. Model checking [2], [3], automated theorem proving [2] and interactive theorem proving [2] techniques have all been used for the verification of digital designs. However, all of these formal verification based approaches do not have a well defined methodology/language to express the specification to facilitate and better communicate between the design and verification processes. Lack of such important facility poses a big challenge for verification engineers in accurately specifying what is to be verified and tested. Moreover, the state-based model checking and the automated theorem proving based techniques cannot cater for large designs due to their inherent state-space explosion problem [4] and the high computational requirement issues [5], [6], respectively. Similarly, the higher-order-logic theorem proving based approaches require a significant amount of manual effort [5] or are limited to a subset of digital designs, like combinational circuits [6].

Assertion Based Verification (ABV) [7] has recently emerged as a promising verification tool for hardware verification as it can cater for most of the above-mentioned limitations. ABV uses well-defined temporal language expressions, called assertions, for the specification of the design requirements in a well-defined way. These assertions can then be checked for a given design using the standard simulation based testing approach. ABV has been known to detect bugs in early stages of design flow, improving observability of design and controllability of verification process [8]. It also helps designers and verification engineers to better understand/communicate during the various design and verification stages [9].

However, writing assertions manually itself is a big challenge. This becomes an impeding factor in the verification process when the specifications to be verified are complex and temporal ones. To overcome this issue, many attempts to generate assertions automatically have been made [8]–[17]. GoldMine [10] is one of the most efficient automatic assertion generation engines. It works with the Verilog register transfer level (RTL) code and its simulated data in VCD format for its data mining algorithms to generate the assertions. But, the quality of assertion generation, using this tool totally depends upon the simulation data provided, which in turn relies upon the quality of the testbench used to simulate the RTL design. In order to enhance the quality of assertion generation for ABV, we propose a simple framework that helps the ABV users to create a useful testbench using an easy to use GUI automatically and then simulate the design in order to get the VCD file that is required by GoldMine. Based on the ideas of randomized testing, we ensure to generate effective assertions that are likely to catch corner cases and not biased by the test engineers opinions and inclinations. For illustration purposes, we present a couple of case studies on the verification of 2-bit Magnitude Comparator and BCD to Excess-3 Code Converter circuits. The verification results for these case studies clearly demonstrate the effectiveness of the proposed approach for an industrial setting.

II. Related Work

As it has been very difficult and tedious job to write the assertions manually, the researchers, in the hardware verification domain, have come up with different methodologies to extract assertions automatically from the hardware designs. Currently two main approaches are being used to automate the assertion extraction process: Simulation based approaches [9], [10], [15]–[17], which require the simulation traces generated by the design simulation to automatically generate assertions, and Specification based approaches [8], [11]–[14], which involve the translation of natural language statements into assertions, which verify the behavior of the candidate design with the specifications defined by the industry standard.

First we focus on the specification based approaches that are used in the hardware domain. Li et al. [11] proposed to generate SystemVerilog (SV) code with assertion support from the Unified Modeling Language (UML) based hardware design description. Schweikert et al. [12] proposed an automatic approach to generate assertions using Sequence Diagrams (SD) to verify RTL designs. The SD can generate messages about the one clock cycle behaviors. Silva et al. [14] proposed a new approach that along with SD employs Finite State Machines (FSMs) in the specification. In this approach, FSMs are modeled with State Chart eXtensible Markup Language (SCXML) and UML sequence diagrams (SD). Lee et al. [13] proposed a Message Sequence Charts (MSCs) based approach to generate SystemVerilog Assertions (SVAs). Almost all of the above-mentioned specification based techniques involve different representation of the given design prior to generate assertions. These representations, using UML, MSCs, SCXML and SD, are not common among hardware engineers and require the user to be aware of their usage. This is why these techniques despite their promising results are not adopted widely in the hardware community. A combination of VCD file format (emulation) and Timing Diagram Markup Language (TDML) format is used in [8] to alleviate the above-mentioned difficulty in the specification based assertion generation approaches. However, this technique requires the user to be familiar with timing diagram standards and related tools.

Now, we describe the existing simulation based approaches: Zhang et al. [9], used exhaustive search based Automatic Test Pattern Generation (ATPG) for assertion generation. They claim to get 100% design space coverage by employing the ATPG algorithm PODEM (Path Oriented Decision Making). However, the scope of this approach is limited to combinational circuits only. Rogin et al. [15], proposed automatic property extraction using testbenches. The method, first, extracts the likely properties using the simulation traces and design signals, and then, the candidate properties are combined and checked considering their temporal dependence using simulation. But this method is highly dependent on simulation patterns. Hangal et al. [16] presented, the IODINE tool, which primarily uses a template based dynamic analysis approach to generate assertions from design simulations. Their set of templates includes one-hot, request-acknowledge and mutual exclusion. Moreover, the clocking and reset scheme for the design has to be specified manually to guide the analyzers. Chang et al. [17] employed a sequential data mining approach to extract hardware assertions. Their technique views assertion extraction as a stand-alone problem and therefore requires the signal selection from the simulation traces in the preprocessing stage by user using domain knowledge for design. Vasudevan et al. [10] combined data mining and static analysis techniques to develop a tool, called GoldMine, to generate assertions from RTL designs and their simulation traces. The users can input their own simulation data/testbenches or the tool generates it using the Data Generator module, which employs its own random testbench generator. However, the testbench, created by the Data Generator module, does not always guarantee to generate meaningful test patterns. This limits the use of GoldMine because the data miner depends upon the simulation traces generated by the Data Generator. Although GoldMine cannot generate unbounded temporal assertions due to the limit put by the simulation but it is found that it can generate assertions with sufficiently high coverage [18]. The tool is entirely automatic and also has been enhanced to evaluate assertions based upon fault coverage [19]. The problem of temporal assertions has been addressed in [20], but their technique still needs improvements as they are unable to verify the liveness property, which involves *eventually* operator.

III. Proposed Methodology

The proposed approach to implement the ABV framework is depicted in Figure 1. The framework has four main blocks: Automatic Testbench Generator, HDL Simulator, Assertion Generation Engine (AGE), and Formal Verifier. Each block depends upon the previous block's output to work in the proposed flow starting with an RTL Verilog code file. The description of each block is given below:

A. Automatic Testbench Generator

The first block of our proposed framework uses an automatic Verilog testbench generator, VerTGen [21], for the given Verilog code. VerTGen is a GUI based automatic random testbench generator tool for Verilog models. It is a user friendly tool that supports all of the major probability distributions to be selected for random pattern generations. It can be used to generate testbenches for both combinational and sequential circuits. In this paper, we extend this tool to make it more useful in our framework by including the feature to provide the Verilog code lines required to dump the simulation data in the VCD format that allows us to work with the Goldmine tool. The lines added in VerTGen generated testbench to dump simulation data has following signature:

```
$dumpfile("moduleName.vcd");
```

In the above statement, *moduleName* represents the name of the module, top module or some submodule, for which the data dumping is to be done in the simulation phase. The name of output VCD file is given the same name as of the module to ease the process in next steps.

```
$dumpvars(0,moduleName_bench.moduleName_);
```

The *moduleName_bench* represents the testbench file name, *moduleName_* is an instantiation of the Verilog module for which we need the variable dumping. This above code line directs the simulator to the variables to be dumped during a running simulation. The naming convention adopted in the VerTGen testbench allows us to automate the next steps.
The output of this block is a randomized Verilog testbench file for the given RTL code.

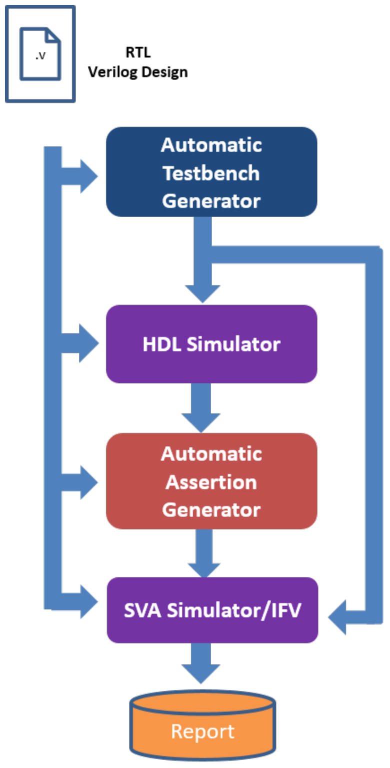


Fig. 1. Proposed Methodology

B. HDL Simulator

The main purpose of this block is to get the initial simulation data required for the GoldMine AGE. GoldMine requires simulation data of the RTL code as a VCD file. This block is basically a Verilog HDL simulator that supports the Verilog simulation data dumping in the VCD file format. It takes two inputs: Verilog RTL design and its testbench. Any Verilog simulator, like ModelSim or Synopsys VCS, that supports the functional simulation of the input RTL code can fit this role. The output of this block is a VCD file containing the simulation data.

C. Assertion Generation Engine (AGE)

This block basically comprises of the GoldMine AGE. This is an automatic assertion generation engine that takes the RTL code along with its simulated data. The GoldMine AGE basic modules include a static analyzer and a data miner for assertion generation. The static analyzer takes RTL code, analyses it for domain specific information and acts as guidance tool for the data miner. GoldMine uses A-Miner as data miner, which takes simulated data in the VCD format. A-Miner works on different supervised learning algorithms and pattern matching to generate assertions. It is important to note that the GoldMine AGE generates SystemVerilog Assertions (SVA) for the input Verilog RTL code that truly depends upon the simulated data in the VCD file. This block outputs assertions and different metrics related to the quality of the generated assertions as well. These assertions may contain some false assertions as the estimation by the data miner can be wrong.

D. Formal Verifier

Since GoldMine generates likely assertions, therefore, some formal verification tool is required to extract the true assertions out of the likely assertions generated by GoldMine. GoldMine itself provides the facility to integrate Cadence IFV to be used for this purpose. The main role of the formal verifier block is to extract true assertions in an iterative manner. It generates counterexamples for all the failing assertions that do not qualify as true assertions. The passed assertions, i.e., the set of true GoldMine generated assertions, are guaranteed to capture functional behavior of the given design.

IV. Case Studies

A. Case Study 1: 2-Bit Magnitude Comparator

In order to explain the flow of the proposed framework, we first present an easy to understand example of a 2-bit magnitude comparator. The block diagram, given in Figure 2, shows that the considered comparator module takes two 2-bit numbers as an input: A1, A0 are most significant bit (MSB) and least significant bit (LSB) of the first number, respectively, and similarly, B1 and B0 represent the MSB and LSB of the second number, respectively. The Verilog code of the comparator is shown in code Listing 1.

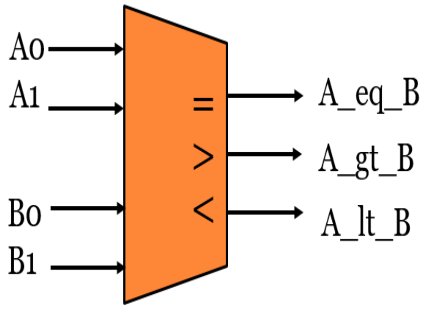


Fig. 2. 2-Bit Magnitude Comparator Block Diagram

Listing 1. Verilog main module

```
//---------------------------------------------
//    Case Study 1: 2-bit Magnitude Comparator
//---------------------------------------------
module Compar2(A_lt_B, A_gt_B,A_eq_B, A1, A0, B1, B0);
```

```
  input A1,A0, B1, B0;
  output A_lt_B, A_gt_B, A_eq_B;

  assign A_lt_B = ({A1,A0} < {B1 ,B0});
  assign A_gt_B = ({A1,A0} > {B1 ,B0});
  assign A_eq_B = ({A1,A0} == {B1,B0});
endmodule
//-------------------------------
```

As stated earlier that one of the distinguishing features of the proposed framework is that it allows the user to start with just the Verilog design file. Starting with the Verilog code, the flow of the proposed framework is explained as the following steps.

1. The first step is to generate the testbench of the given Verilog design using the VerTGen GUI. This step generates a randomized testbench with many flexible user defined options, like random distribution selection, seed value and input variable range assignment etc. Also the VerTGen generated testbench contains the necessary code lines to dump the simulation data in the required VCD format. The testbench generated by VerTGen is given in Listing 2.

Listing 2. Testbench for 2-bit Magnitude Comparator

```
//---------------------------------------------
//    Testbench for 2-bit Magnitude Comparator
//---------------------------------------------
'define SEED_INIT 1510035665
module comparator_2_bench;
reg A1,A0, B1, B0;
wire A_lt_B, A_gt_B, A_eq_B;
reg [31:0] seed;
integer in, out, r;

comparator_2 comparator_2_(A_lt_B, A_gt_B,A_eq_B, A1, A0,
     B1, B0);
integer i_loop;
initial begin
in  = $fopen("seed.txt","r");
if(in==0)
seed='SEED_INIT;
else begin
r={$fscanf(in," %b\n",seed)};
$fclose(in);
end
for( i_loop = 0; i_loop < 10000; i_loop = i_loop +1) begin
#5  A1 = {$random(seed)};
#5 A0 = {$random(seed)};
#5  B1 = {$random(seed)};
#5  B0 = {$random(seed)};
#10;
end
out = $fopen("seed.txt","w");
$fwrite(out," %b\n",seed);
$fclose(out);
end
initial begin
#10;
$display("Simulation Result are as follows:");
$monitor("A_lt_B: %b , A_gt_B: %b ,A_eq_B: %b , A1: %b ,
     A0: %b , B1: %b , B0 : %b ",A_lt_B, A_gt_B,A_eq_B, A1
     , A0, B1, B0);
end
initial begin
$dumpfile("<PATH TO SAVE>/comparator_2.vcd");
$dumpvars(0, comparator_2_bench.comparator_2_);
#100000 $finish;
end
endmodule
//---------------------------------------------
```

2. The next step is to simulate the Verilog design using the testbench generated in the previous step. We used the ModelSim Starter version for our example. Upon successful simulation, we get the VCD file containing all the simulated data.
3. Now, we run the assertion generation tool of Goldmine on the files generated in the last step. Upon successful execution, the invoked tool generates the required assertions and its rank related metrics. In this case study, the tool generated a total of 24 assertions for variables *A_lt_B*, *A_gt_B* and *A_eq_B*. For each variable, it generated 8 likely assertions. For the sake of brevity, we have shown the generated assertions for *A_eq_B* only in Listing 3.
4. To extract the true assertions from the set of likely assertions, the next step is to pass them to the formal verifier. GoldMine allows us to modify its configuration file to set the path and license information for Cadence IFV so that it can extract the true assertions automatically after the previous steps. After the path and license information is set in the GoldMine configuration file, then the assertions that pass the test are labeled as verified assertions, and the ones that fail are reused for counterexample generation.

Listing 3. Assertions by GoldMine

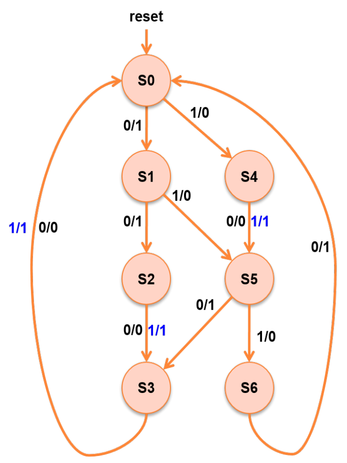
```
//---------------------------------------------
//    Case Study 1: Assertions generated for A_eq_B
//---------------------------------------------
reg clk;
always @ (posedge clk)
begin
gm0 : assert property ( ( A0 == 1 && B0 == 0 ) |-> (
     A_eq_B == 0 ) );
gm1 : assert property ( ( A0 == 0 && B0 == 1 ) |-> (
     A_eq_B == 0 ) );
gm2 : assert property ( ( A1 == 1 && B1 == 0 && B0 == 0 )
     |-> ( A_eq_B == 0 ) );
gm3 : assert property ( ( A1 == 1 && B1 == 0 && B0 == 1 )
     |-> ( A_eq_B == 0 ) );
gm4 : assert property ( ( A1 == 0 && B1 == 1 ) |-> (
     A_eq_B == 0 ) );
gm5 : assert property ( ( A1 == 0 && A0 == 0 && B1 == 0 &&
      B0 == 0 ) |-> ( A_eq_B == 1 ) );
gm6 : assert property ( ( A1 == 0 && A0 == 1 && B1 == 0 &&
      B0 == 1 ) |-> ( A_eq_B == 1 ) );
gm7 : assert property ( ( A1 == 1 && A0 == 0 && B1 == 1 &&
      B0 == 0 ) |-> ( A_eq_B == 1 ) );
gm8 : assert property ( ( A1 == 1 && A0 == 1 && B1 == 1 &&
      B0 == 1 ) |-> ( A_eq_B == 1 ) );
end
//------------------------------
```

B. *Case Study 2: BCD to Excess-3 Code Converter*

This case study showcases the assertion generation for sequential circuits by GoldMine using the proposed framework. The circuit considered here is a binary-coded decimal (BCD) to Excess-3 code converter using Mealy FSM. It is a serial converter, which takes four bits of BCD code from LSB to MSB serially and returns the corresponding four bits of Excess-3 code. The conversion table and the state diagram of FSM representing the behavior of BCD to Excess-3 code converter are given in Figure 3.

Decimal Digit	BCD Code	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

(a) Conversion Table



(b) State Diagram of FSM

Fig. 3. BCD to Excess-3 Code Converter: Conversion Table and FSM

The Verilog RTL code for this case study is provided in Listing 4. The input *x* designates the bit of the BCD code and the corresponding output to that bit is given out by variable *x*. The variable *pstate* represents the present state of the FSM. The FSM has seven states, designated from *S0* to *S6*, as depicted in Figure 3(b).

Listing 4. BCD to Excess-3 Code

```
//-----------------------------------------
//    Case Study 2: BCD to Excess-3
//-----------------------------------------
module BCD_to_Excess_3 (y, x, pstate, clk, reset);
output y, pstate;
input x, clk, reset;
parameter  S0 = 3'd0,  S1 = 3'd1,  S2 = 3'd2,
S3 = 3'd3,  S4 = 3'd4, S5 = 3'd5,
S6 = 3'd6;
reg[2: 0] pstate, nxtstate;
reg y;
always @ (posedge clk or negedge reset)
if (reset== 0) pstate  <=  S0;
else pstate  <=  nxtstate;
always @ (pstate or x)
begin
case (pstate)
S0: if (x)
begin  nxtstate = S4; y = 0;  end
else
begin  nxtstate = S1; y = 1;  end
S1: if (x)
begin  nxtstate = S5; y = 0;  end
else
begin  nxtstate = S2; y = 1;  end
S2:  begin     nxtstate = S3; y = x;  end
S3:      begin  nxtstate = S0; y = x;  end
S4:      begin  nxtstate = S5; y = x;  end
S5: if (x)
begin  nxtstate = S6; y = 0;  end
else
begin  nxtstate = S3; y = 1;  end
S6:     begin   nxtstate = S0; y = !x; end
default:       begin nxtstate = 3'dx; y = x; end
endcase
end
endmodule
//------------------------------
```

Following similar steps as done in Case Study 1, the framework generated assertions for output signal *y* of the RTL code are listed in Listing 5. GoldMine generated a total of 22 assertions for this case. It can be seen that the assertions generated in this case study are complex and have clock cycles information related to signal values. For example, consider assertion *gm0* in the Listing 5. This assertion says that if the value of *pstate* is 010 (*S2*) or 000 (*S0*) and *x=1*, and after 2 clock cycles if *x=1* then the output bit *y* will be 0. This can be compared from the state diagram of the FSM in Figure 3(b) for illustration purposes.

Listing 5. Assertions by GoldMine

```
//-----------------------------------------------
//    Case Study 2: Assertions for output y
//-----------------------------------------------
always @ (posedge clk)
begin
no_reset: assume property ( reset == 1 );
gm0 : assert property ( ( pstate[2] == 0 && pstate[0] == 0
     && x == 1 ) ##2 ( x == 1 ) |-> ( y == 0 ) );
gm1 : assert property ( ( pstate[2] == 1 && pstate[0] == 1
     ) ##2 ( x == 1 ) |-> ( y == 0 ) );
gm2 : assert property ( ( pstate[1] == 1 ) ##1 ( x == 0 )
    ##1 ( x == 1 ) |-> ( y == 0 ) );
gm3 : assert property ( ( pstate[1] == 0 && x == 1 ) ##1 (
     x == 1 ) ##1 ( x == 1 ) |-> ( y == 0 ) );
gm4 : assert property ( ( pstate[2] == 0 && pstate[1] == 0
     && pstate[0] == 1 && x == 0 ) ##2 ( x == 0 ) |-> ( y
     == 0 ) );
gm5 : assert property ( ( pstate[1] == 0 && pstate[0] == 0
     && x == 0 ) ##1 ( x == 0 ) ##1 ( x == 0 ) |-> ( y ==
     0 ) );
gm6 : assert property ( ( pstate[1] == 1 && pstate[0] == 1
     ) ##1 ( x == 1 ) ##1 ( x == 0 ) |-> ( y == 0 ) );
gm7 : assert property ( ( pstate[2] == 1 && pstate[1] == 0
     && pstate[0] == 0 ) ##1 ( x == 1 ) ##1 ( x == 1 )
    |-> ( y == 0 ) );
gm8 : assert property ( ( pstate[2] == 0 && pstate[1] == 1
     && pstate[0] == 0 ) ##1 ( x == 1 ) ##1 ( x == 1 )
    |-> ( y == 0 ) );
gm9 : assert property ( ( pstate[2] == 1 && pstate[0] == 0
     && x == 1 ) ##1 ( x == 1 ) ##1 ( x == 0 ) |-> ( y ==
     0 ) );
gm10 : assert property ( ( pstate[2] == 0 && pstate[0] ==
    0 && x == 1 ) ##2 ( x == 0 ) |-> ( y == 1 ) );
gm11 : assert property ( ( pstate[2] == 1 && pstate[0] ==
    1 ) ##2 ( x == 0 ) |-> ( y == 1 ) );
gm12 : assert property ( ( pstate[1] == 1 ) ##1 ( x == 0 )
     ##1 ( x == 0 ) |-> ( y == 1 ) );
gm13 : assert property ( ( pstate[1] == 0 && pstate[0] ==
    0 && x == 0 ) ##1 ( x == 1 ) ##1 ( x == 0 ) |-> ( y
    == 1 ) );
gm14 : assert property ( ( pstate[1] == 1 && pstate[0] ==
    1 ) ##1 ( x == 1 ) ##1 ( x == 1 ) |-> ( y == 1 ) );
gm15 : assert property ( ( pstate[2] == 0 && pstate[1] ==
    0 && pstate[0] == 1 && x == 0 ) ##2 ( x == 1 ) |-> (
    y == 1 ) );
gm16 : assert property ( ( pstate[1] == 0 && pstate[0] ==
    0 && x == 0 ) ##1 ( x == 0 ) ##1 ( x == 1 ) |-> ( y
    == 1 ) );
gm17 : assert property ( ( pstate[1] == 0 && pstate[0] ==
    0 ) ##1 ( x == 1 ) ##1 ( x == 0 ) |-> ( y == 1 ) );
gm18 : assert property ( ( pstate[2] == 0 && pstate[0] ==
    0 && x == 0 ) ##1 ( x == 1 ) ##1 ( x == 0 ) |-> ( y
    == 1 ) );
gm19 : assert property ( ( pstate[1] == 0 && x == 1 ) ##1
    ( x == 1 ) ##1 ( x == 0 ) |-> ( y == 1 ) );
gm20 : assert property ( ( pstate[2] == 1 && pstate[1] ==
    1 ) ##1 ( x == 1 ) ##1 ( x == 1 ) |-> ( y == 1 ) );
gm21 : assert property ( ( pstate[2] == 0 && pstate[1] ==
    0 ) ##1 ( x == 0 ) ##1 ( x == 1 ) |-> ( y == 1 ) );
end
//-------------------------------
```

V. Conclusions

In this paper, we presented a semi-automatic hardware verification framework, which employs randomized testing along with assertions to get a better insight into the design and its bugs. The proposed methodology uses VerTGen for randomized testbench generation and GoldMine for automatic assertion generation for a given RTL circuit. The methodology is easy to adopt and time saving due to its automatic nature. In this work, we have also showcased the ability of the framework to generate assertions effectively for both combinational and sequential circuits.

References

[1] T. Coe, "Inside the pentium-fdiv bug," *DR DOBBS JOURNAL*, vol. 20, no. 4, p. 129, 1995.

[2] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology, Third Edition*, pp. 7162–7170, IGI Global, 2015.

[3] B. Bèrard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[4] C. Baier, J.-P. Katoen, *et al.*, *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[5] W. Bibel, *Automated theorem proving*. Springer Science & Business Media, 2013.

[6] S. Shiraz and O. Hasan, "A hol library for hardware verification using theorem proving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[7] K. Datta and P. Das, "Assertion based verification using hdvl," in *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 319–325, IEEE, 2004.

[8] M. O. Kayed, M. Abdelsalam, and R. Guindi, "Synthesizable sva protocol checker generation methodology based on tdml and vcd file formats," in *High Level Design Validation and Test Workshop (HLDVT), 2016 IEEE International*, pp. 1–8, IEEE, 2016.

[9] T. Zhang, D. Saab, and J. A. Abraham, "Automatic assertion generation for simulation, formal verification and emulation," in *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*, pp. 471–476, IEEE, 2017.

[10] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 626–629, March 2010.

[11] L. Li, F. P. Coyle, and M. A. Thornton, "Uml to systemverilog synthesis for embedded system models with support for assertion generation," in *Proceedings of the ECSI forum on design languages*, 2007.

[12] M. Schweikert, T. Dornes, and H. Eveking, "Using sequence diagrams to specify and to generate rtl assertions," in *Proceedings of the Fifth International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS'ii)*, 2011.

[13] P. S. Lee and I. G. Harris, "Message sequence charts for assertion-based verification," *CECS Technical Report*, 2013.

[14] W. Silva, E. Bezerra, M. Winterholer, and D. Lettnin, "Automatic property generation for formal verification applied to hdl-based design of an on-board computer for space applications," in *Test Workshop (LATW), 2013 14th Latin American*, pp. 1–6, IEEE, 2013.

[15] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Automatic generation of complex properties for hardware designs," in *Design, Automation and Test in Europe, 2008. DATE'08*, pp. 545–548, IEEE, 2008.

[16] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "Iodine: a tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of the 42nd annual Design Automation Conference*, pp. 775–778, ACM, 2005.

[17] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pp. 607–612, IEEE Press, 2010.

[18] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, 2013.

[19] V. Athavale, "Coverage analysis for assertions and emulation based verification," *Master's Thesis, University of Illinois at Urbana-Champaign*, 2012.

[20] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 67–72, EDA Consortium, 2015.

[21] S. A. Murtza, O. Hasan, and K. Saghar, "Vertgen: An automatic verilog testbench generator for generic circuits," in *2016 International Conference on Emerging Technologies (ICET)*, pp. 1–5, Oct 2016.