Effective Error Diagnosis for RTL Designs in HDLs

Tai-Ying Jiang
Dept. of Electronics Engineering
National Chiao Tung University
Hsinchu, Taiwan, ROC
giani@eda.ee.nctu.edu.tw

Chien-Nan Jimmy Liu
Dept. of Electrical Engineering
National Central University
Chung-Li, Taiwan, R.O.C.
jimmy@ee.ncu.edu.tw

Jing-Yang Jou
Dept. of Electronics Engineering
National Chiao Tung University
Hsinchu, Taiwan, ROC
jyjou@ee.nctu.edu.tw

Abstract

We propose an effective approach to diagnose multiple design errors in HDL designs with only one erroneous test case. Error candidates will be greatly reduced while ensuring that true erroneous statements are included in. The probability of correctness for each potential erroneous statement will be estimated such that the most suspected statements are reported first. Experiments show that the size of error candidates is indeed small and the estimation for the probability of correctness for potential error candidates is accurate.

1. Introduction

With the increasing complexity of VLSI circuit designs, a typical design cycle is often split into various stages such that functional mismatches between adjacent design stages often occur. Once a verification tool finds that the design in the current stage (the implementation) does not agree with that in the previous stage (the specification), design error diagnosis and correction is required. Traditionally, in the problem of design error diagnosis, the implementation is often represented as gate level or lower level circuits and the specification is defined as gate-level or higher level circuits. Those methods can be roughly divided into two categories: simulation-based approaches and symbolic approaches.

In the simulation-based approaches, we have to derive a number of erroneous vectors first, which are the vectors that can differentiate the implementation and the specification. By simulating those vectors, the possible error candidates can be trimmed down gradually. In the literature, there are two primary approaches to narrow down the error candidates. Some of them rely on design error models [3,9,10,11], which may include gate errors (missing gate, extra gate, wrong logical connective,…) and line errors (missing line, extra line,…), to rectify a circuit. The others are structure-based methods [2,5,6,7,12], which do not require design error models. By using the structural analysis techniques [2] and performing incremental re-synthesis, the design error can be diagnosed with minimum efforts.

On the other hand, symbolic approaches handle the design error diagnosis problems by using Boolean function manipulation [4,8,13]. In those approaches, Ordered Binary Decision Diagram (OBDD) is used to formulate the necessary and sufficient condition of fixing a single error. Recently, symbolic approaches are extended to handle multiple design errors [13]. As compared to the simulation-based approaches, the advantages of those methods are more accurate and do not require design error models, however, they suffer the memory explosion problem in handling large circuits.

In modern design process, most design errors occur in the early stage of describing the functional behavior of a design in HDLs and design error diagnosis at this stage is often performed by tracing the code manually. However, for modern designs with thousands of lines of HDL code, debugging such circuits manually is a difficult task. Therefore, automatic design error diagnosis techniques for HDL designs are proposed [14,16]. In [14], Vamsi Boppana et al exploits hierarchy available in RTL designs to locate design errors and the information from the simulation of Xlists[15] to capture the effects of design errors within components of RTL designs. Maisaa Khalil et al [16] proposed an approach to point out the the exact or likely error location with the assumption that both the set of test cases and the corresponding simulation results are available. However, the number of the error candidates may still be too large for designers to debug. Furthermore, assuming multiple erroneous test cases being available is impractical. In most cases, designers conduct the job with only one erroneous test case.

In this paper, we propose an error-model independent approach for assisting design error diagnosis for HDL designs, which can handle multiple design errors with only one erroneous test case effectively. We will reduce the number of error candidates while ensuring true errors are included in. The probability of correctness for each potential error candidate is estimated with some heuristics. According

Proceedings of the 11 th Asian Test Symposium(ATS'02)
1081-7735/02 $17.00 © 2002 IEEE

to the estimated probability, the most suspected error candidates will be reported first in the display such that the efforts of debugging can be further reduced.

The remainder of the paper is organized as follows. Section 2 gives the work overview. Section 3 describes the reduction of error candidates. In section 4, we estimate the probability of correctness for each potential error candidate. Finally, we present the experimental results in Section 5, and conclude this paper in Section 6.

2. Work overview

Given a synchronous digital HDL implementation, a specification, which is given as the expected values of all POs and registers at all clock cycles, and one test pattern, which are given by designers to verify the design, we are going to solve the design error diagnosis for HDL design. We check the simulation values of all POs and registers after each simulation clock cycle. If mismatches between the simulation values and the expected ones occur, we take Erroneous Primary Outputs (EPOs) and the faulty HDL design as inputs and output the set of error candidates in an order, which is from the most suspected one to the most innocent one. If not, for the need of our following operations, we will collect the execution statistics of each statement and each PO with correct simulation value at this clock cycle With the collected information, the probability of correctness for each potential error candidate can be estimated. Then, we apply the next set of input vectors for the next clock cycle and continue the simulation until at least one error occurs. If, at the end, there is no mismatch, error may still exist in the design. More simulation or other verification work has to be done. However, that is beyond our scope. The overall flow of our approach is shown in **Figure 1.**

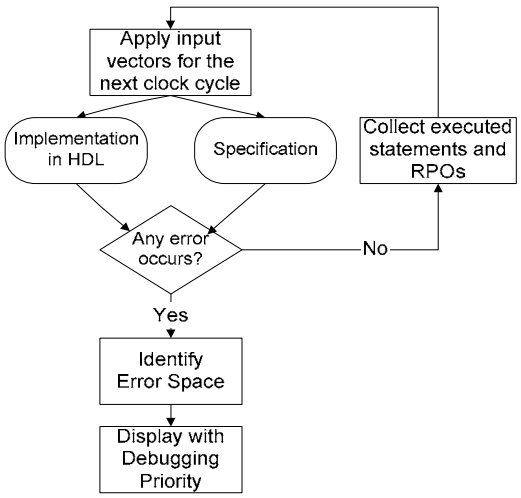

Figure 1. An overall flow of our approach

3. Error space identification

Error space is a set of error candidates for designers to identify their design errors. If debugging is considered without any aid, designers have to find the design errors in the whole HDL design. In other words, the *error space* is the whole HDL description. Therefore, our goal is to minimize the size of *error space* by effectively using the information collected during the simulation session while ensuring that the true design errors are always included in *error space*. The reduction of *error space* can be very helpful because its size directly corresponds to the efforts of debugging.

In this section, we first find initial error candidates in section 3.1. In section 3.2, we analyze data dependency of EPOs and reduce the size of *error space* further. Finally, we present the overall algorithm of *error space* identification in Section 3.3.

3.1 Executed statements of EOC

Definition 1: Executed Statements of Error-occurring Clock Cycle (ES(EOC)) are the Executed Statements of the Clock Cycle, in which an error appears for the first time.

As illustrated in Fig. 1, we check the values of all POs and registers so that erroneous effects can not be propagated to the next clock cycle. Therefore, the potential error candidates can be limited in the ES(EOC). Therefore, we eliminate those statements, which are not executed in this clock cycle, from the error candidates.

In order to explain ES(EOC) more clearly, we use the Verilog code shown in Fig. 2 as an example. Assume that the code in Fig. 2 is the correct design that designers expect. However, for some reasons, the statement s9 is written incorrectly and becomes "*w2 = PI4*". The applied input vectors for each time instance and the corresponding values of POs are shown in Fig. 3. Because an error occurs at PO1 at 25ns, EOC is the clock cycle, which is from time=15ns to time=25ns. At time=20ns, *s1, s2, s4,* and *event2* are triggered because of the value changes of PI1 and PI4. Since *sel2=1'b0,* the execution statistics of statements under the evnet control of *event2* is that *dec.2* (decision or conditional statement) and *s9* are executed. E*vent1* is triggered due to the rising edge of *CLK* at 25ns. Because the *event1* is triggered and sel1=1'b1, *dec.1* and *s6* are executed. Therefore, ES(EOC) are *s1*, s*2*, s4, s6, s9, e*vent1*, d*ec.1*, *event2, and dec.2*. Note that the error source s*9* is included in ES(EOC).

3.2 Relation space extraction

Definition 2 : Relation Space (RS) of a specific primary output POi*,* which is denoted as RS(POi), is a set of

statements that are related to POi in the data flow graph of the HDL design.

According to the definition above, only the statements in RS(EPOi) have influence on the value of EPOi. Statements, which are not in RS(EPOi), are impossible to be the error sources and can be eliminated from *error space*.

We will show an example of extracting RS(EPO1) in the control data flow graph with the inputs of ES(EOC), which are *s1*, s2, s4, s6, s9, e*vent1*, d*ec.1*, *event2, and dec.2*. First, the control data flow graph (CDFG), which can be obtained by analyzing the data dependency of the HDL code, is built as shown in Fig. 4, where *s* denotes a statement and *dec.* represents a conditional statement or a decision. With the CDFG, we can obtain RS(EPO1) by conducting a back trace from PO1 to the PIs according to the relationship in the data flow. When we look over the HDL code shown in Fig. 2 and the CDFG shown in the Fig. 4, the first node we meet is *dec.1*. In ES(EOC), only *s6* is the driving statement of EPO1. We add it in RS(EPO1) and then find the driving statements of *s6* and *dec.1*. The driving statements of *s6* are *dec.2* and *s9* and the driving statements of *dec.1* is *s1*. We add them in RS(EPO1), too. Similarly, we find the driving statements of *dec.2* and *s9* and add them into RS(EPO1). Finally, RS(EPO1) includes the statements { *dec.1, s6, s1, event1, dec.2, s9, s2, event2* }. *Event1* and *event2* are added in RS(EPO1) because they are event controls of statements mentioned above.

```
module exm(PO2,PO1,PI1,PI2,PI3,PI4,clk);
input PI1,PI2,PI3,PI4,clk;
output PO2,PO1;
reg PO1,w2;
wire sel1,sel2,w1;
assign sel1 = PI1 & PI2;                s1
assign sel2 = PI3 | PI4 ;               s2
assign w1 = PI2 ^ PI3 ;                 s3
assign PO2 = w2 | PI1;                  s4
always @ ( posedge clk )                event1
begin
case( sel1 )                            dec.1
1'b0:   PO1 = w1;                       s5
1'b1:   PO1 = w2;                       s6
default :   PO1 = w1;                   s7
endcase
end
always @ (sel2,PI1,PI3,PI4)             event2
begin
if( sel2 )                              dec.2
w2 = PI3;                               s8
else
w2 = PI4 | PI1;                         s9
end
endmodule
```

Figure 2. An example written in Verilog HDL

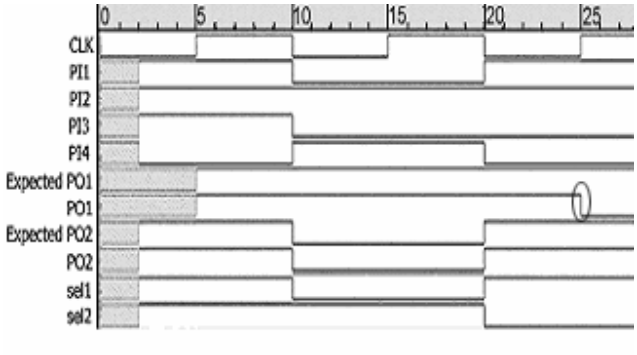

Figure 3. The waveform of signals in Verilog code shown in **Figure 2**

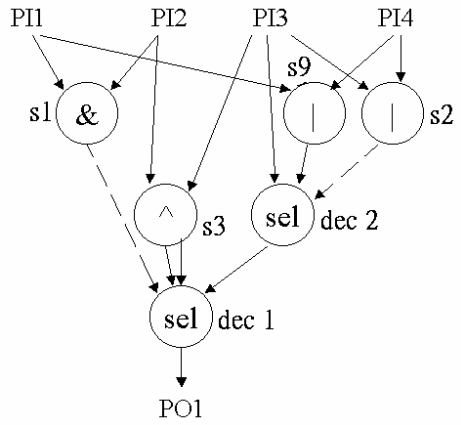

Figure 4. The Control Data Flow Graph (CDFG) of EPO1

3.3 Error space identification

The whole algorithm of our *error space* identification is shown as the pseudo code in Fig. 5. The *error space* identification procedure takes EPOs, the HDL design, and ES(EOC) as its inputs and output *error space* by eliminating impossible error candidates. At the beginning, the set ES (Error Space) and the set ES_highpriority are both empty. The function of Find_ES is to extracts RS(EPOi) in ES(EOC). The whole *error space* identification eliminates impossible error candidates by taking the intersection of ES(EOC) and each RS(EPOi). The resulted set of statements is added into ES. The union of these statements sets obtained by Find_ES is *error space*. Therefore, only one erroneous case is needed while applying our approach.

In our algorithm, we will keep the statements that appear in more than one RS(EPOi) in the set ES_highpriority. Because these statements are related to more than one EPO, intuitively, they are more possibly erroneous and are prioritized to be checked first while tracing design errors. This is done for reporting statements in *error space* with *debugging priority, which will be* discussed in section 4,

such that the most suspected statements are reported first and vice versa.

```
Error_Space_Identifying (EPOs, the HDL design, ES(EOC) )
{
  ES ← 0 ; // ES is error space
  ES_highpriority ← 0 ; // Statements in ES with high
                        //debugging priority
  For ( each EPOi )
  {
    S ← Find_ES ( EPOi, ES(EOC) );
    // Find RS (EPOi) with ES (EOC) ;
    ES_highpriority ← ES_highpriority ⋃ ( ES ⋂ S ) ;
    ES ← ES ⋃ S;
  }
}
```

Figure 5. The pseudo code of *error space* identification

4. Debugging priority

Instead of further reducing the size by using more complicated methods, we plan to display the statements in *error space* with a priority, which is called *debugging priority*, such that the most suspected statements are reported first. By estimating the probability of correctness for all statements in *error space* with *confidence scores calculation*, *debugging priority* can be calculated for debugging purpose.

4.1 Confidence scores calculation

Due to the control of conditional statements, only a part of statements in RS(POi) can really affect the value of POi at a specific time. Those statements are called the *sensitized statements* of the POi. For example, as shown in Fig. 6, the evaluation result of the decision "*if(sel1)…else…*" is "*TRUE*" and the evaluation result of the decision "*if(sel2)…else…*" is "*FALSE*". Therefore, only statements *f2* and *f4* are possible to affect the value of PO1 and are also observed by PO1. These two statements *f2* and *f4* are defined as the *sensitized statements* of PO1 (SS(PO1)).

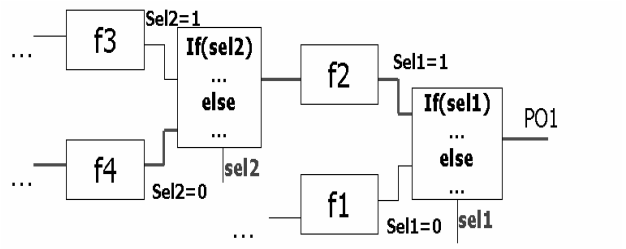

Figure 6. An example of *sensitized statements (SS)*

According to the above definition, only *sensitized statements* are able to affect the values of their corresponding POs. Although the consistence between expected values and simulation values of POs and registers does not imply the correctness of *sensitized statements*, it may still provide some degree of confidence level for the correctness of SS. Thus, the formula of *confidence score* is that each statement in SS(RPOi) will be given one point.

To explain this, we first consider two situations that there is at least one error in SS while the simulation value of PO is correct. These two situations are "non-activated" errors and masked errors. Errors are said "not-activated" because the applied input pattern can not differentiate the outputs of the erroneous statements and the correct ones. The probability *P(non-activated)* is generally very small. For example, if the correct statement is "*assign c = a + b;*" and the erroneous one is "*assign c= a * b*", only applying the input patterns "*a=2; b=2;"* and *"a=0; b=0;"* may generate the same outputs for both statements. Otherwise, errors are activated.

Even if the errors are activated, their syndromes may still be masked by the succeeding statements such that the values of POs are the same as expected. Consider a simple example shown in Fig. 7. The applied input vector is "*PI1=2'b10; PI2=2'b01;"* and the values of all variables are *"E=2'b10; sel=1'b1; B=2'b11; D=2'b11; C=2'b10; A=2'b10; PO1=2'b01;"*. If the statement *f1* becomes an erroneous statement "*D=PI1*;", the value of *D* will become *2'b10* instead of *2'b11*. However, the output of the statement *f2* is still *C=2'b10*. There is no syndrome shown at *PO1* because the activated error is masked by the statement *f2*. The probability *P(mask|activated)* is not high in general.

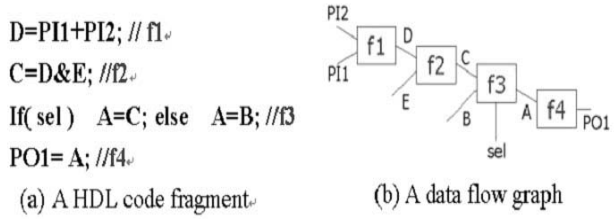

Figure 7. An example of masked errors

Given *P(non-activated)* and *P(mask|activated)*, the possibility for the sensitized statements to be erroneous sources while their corresponding PO is correct, which is denoted as *P(error|CPOi)*, can be estimated as

$$P(error|CPOi) = P(non\text{-}activated) + P(activated) * P(mask|activated)$$

Since *P(non-activated)* and *P(mask|activated)* are generally not high, *P(error|CPOi)* is generally not high, either. For instance, if *P(non-activated)*=0.1 and *P(mask|activated)*=0.3, *P(error|CPOi)*=0.1+0.9*0.3=0.37.

For each PO with correct simulation value at each simulation cycle, the SS(CPOi) will be given one point because P(error|CPOi) is generally not high. If a statement gets 5 points, which is denoted as P(error|5CPOs), the probability for it to be erroneous can be estimated as

$$P(error|5CPOs)=P(error|CPO_1)P(error|CPO_2)P(error|CPO_3)P(error|CPO_4)P(error|CPO_5)$$

Assume that each event of the probability is roughly independent to each other. If we take the value of *P(error|CPOi)* calculated previously for each *P(error|CPOi), P(error|5 CPOs)* can be roughly estimated as $0.37^5 = 0.007$. Therefore, **the more points a statement has; the less possible it is to be a design error**. The formula of *confidence score* is feasible and suitable capatable to represent the confidence level of each statement in *error space*.

Only the executed statistics of statements and CPOs of the clock cycles before EOC are needed to obtain SS(CPOi) and to calculate *confidence score*. The information is collected before any error occurs in this erroneous test case. Therefore, displaying statements in *error space* with *debugging priority* need only one erroneous test case. Generating another erroneous test case to debug is optional.

4.2 An example of debugging priority

In order to demonstrate how we calculate *confidence scores* and display statements in *error space* with *debugging priority*, we continue the example shown in Fig. 2. We find that EOC is the clock cycle, from time=15ns to time=25ns, and *error space* is {*s1, s2, s6, s9, event1, dec.1, event2, dec.2*}.

After using all CPOs in the simulation cycle before EOC and EOC, *confidence scores* are calculated. We display these statements with *debugging priority* according to the *confidence score*s as shown in Fig. 8. A statement with less score is displayed first for its high probability to be erroneous and vise versa. The number in the round brackets ( ) after each statement is their *confidence scores*.

In the above example, we can see that the error source *s9* is given one point only and displayed in the first line. Therefore, although the number of statements in the *error space* is eight, users can find their design error at the first line in the display if they trace the statements according to *debugging priority.*

```
S9 (1) :        w2 = PI4;
S6 (1) :        1'b1:   PO1 = w2;
S1 (2) :        assign sel1 = PI1 & PI2;
Dec.1 (2):      case( sel1 ).....endcase
Event1 (2):     always @ ( posedge clk )
Dec2. (4):      if( sel2 )...else...
Event2 (4):     always @ (sel2 or PI1 or PI3 or PI4)
S2 (4):         assign sel2 = PI3 | PI4 ;
```

Figure 8. The report with *debugging priority*

5. Experimental Results

In this section, we will show the experimental results on five designs written in Verilog HDL. The design *Matrix2X2* is a design for the 2x2 matrix multiplication. The design *FSM* is a simple mealy finite state machine used to control traffic lights. The design *BlackJack* is the controller for black jack game machine. The design *ADPLL_CTRL* is the controller of an all-digital phase lock loop design. The design *PCPU* is a simple 32-bit pipelined DLX CPU.

For every design, we first simulate it once to obtain the correct variable-dump file. Then, we change two statements in the design to create erroneous source. With the created errors, we run our program on a 300MHz UltraSparc II workstation. The results are shown in Table 4. The column "*Lines"* gives the number of lines in the HDL design. The average number of statements in *error space* is recorded in the column "*AVG # lines in error space*". Total experimental cases for each design are shown in the column "*# total cases*". In the column "*# cases*", we give the number of cases that the true erroneous source appears for each period in the displayed list of *error space* to show how effective *debugging priority* works. For example, in the row "Matrix2X2", the sixteen in the column "*0%~20%*" represents that there are 16 experimental cases in which true erroneous sources appear in the first twenty percent in the displayed list of *error space* and 4 cases in which true erroneous sources appear within the period of the twenty percent to fifty percent.

The efforts of debugging directly correspond to the number of error candidates. With the aid of *error space* identification, the size of error candidates is reduced from the whole design to *error space*. Therefore, we define the efficiency of *error space* identification (Eff. of ESID) as "*Lines"* divided by "*AVG # lines of error space"*. In the column "*Eff. of ESID*", we can observe that "*Eff. of ESID*" is high and the debugging efforts are indeed reduced with the help of *error space* identification. Besides *error space identification*, the additional help of *debugging priority* is effective, too. We can tell from the great reduction of the effective average number of statements in *error space* with

debugging priority (AVG_#ES_ESID+DP), which can be estimated as

$$AVG\_\#\_ES \times \left(\frac{0+0.2}{2} \times \frac{\#cases\_0-20\%}{\#total\_cases} + \frac{0.2+0.5}{2} \times \frac{\#cases\_20-50\%}{\#total\_cases} + \frac{0.5+1}{2} \times \frac{\#cases\_50\%-100\%}{\#total\_case} \right)$$

Take the row of FSM as an example. AVG_#ES_ESID+DP of FSM is

$$9.5*(0.1*17/20+0.35*3/20)=1.31.$$

The efficiency of error space identification with additional help of debugging priority (Eff. of ESID+DP) can be calculated as "Lines" divided by "AVG_#ES_ESID+DP". From the column "Eff. of ESID+DP", we know that the debugging efforts are greatly reduced with the aid of our tool and the estimation of the probability of correctness for each potential error candidate is accurate.

Design	Lines	# lines of *EP* AVG/Max/Min	# cases			Eff. of ESID	Eff. of ESID +DP	# total cases
			0~20%	20~50%	50%~			
Matrix 2X2	80	5.3 / 7 / 3	16	4	0	15.1	100.7	20
FSM	113	9.5 / 16 / 7	17	3	0	11.9	86.3	20
Black Jack	195	10.1 / 21 / 3	16	2	2	19.3	101.6	20
ADPLL CTRL	352	21.5 / 42 / 4	13	6	1	16.4	78.9	20
PCPU	952	26.3 / 53 / 7	16	2	2	36.2	190.5	20

Table 4. Experimental results of the design error diagnosis

6. Conclusion

An effective approach for automatic design error diagnosis to diagnose multiple errors in HDL designs with only one erroneous test case is proposed. For the error candidates, we will first eliminate some impossible statements by taking the intersection of ES(EOC) and the relation space extraction. The estimation of the probability of correctness for each potential error candidates in *error space* is conducted by calculating *confidence scores*. The experimental results show that *error space* is indeed small and true erroneous statements are included in and *confidence score* generally responds to the correctness of the statement. Therefore, the effective size of *error space* can be considered smaller than the original one.

7. References

[1] E. J. Aas, K. Klingsheim, and T. Steen, "Quantifying design quality: A model and design experiments", in Proceeding EURO-ASIC, 1992, pp. 172-177.

[2] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation", in IEEE transactions on CAD, 7(1): 138-148, January 1988.

[3] P. Y. Chung, Y. M. Wang, and I. N. Hajj, "Diagnosis and correction of logic design errors in digital circuits", in Proceeding Design Automation Conference, 1993, pp. 503-508.

[4] D. Brand, "Incremental synthesis", in Proc. of the Intl. Conference on Computer Aided Design, 1992, pp. 126-129.

[5] S.Y. Huang, K. T. Cheng, K. C. Chen, and D. I. Cheng, "Error tracer: a fault simulation-based approach to design error diagnosis", in IEEE Intl. Test Conference, 1997, pp. 974-981.

[6] H. T. Liaw, J. H. Tsaih, and C. S. Lin, "Efficient automatic diagnosis of digital circuits", in Proceeding Intl. Conference on Computer Aided Design, 1990, pp. 464-471.

[7] M. Tomita, and H. H. Jiang, "An algorithm for locating logic design errors", in Proceeding Intl. Conference Computer-Aided Design, 1990, pp. 468-171.

[8] [8] M. Tomita, T. Yamamoto, F. Sumikawa and K. Hirano, "Rectification of multiple logic design errors", in Proceeding of ACM/IEEE DAC, 1994, pp. 212-217.

[9] [9] S. Y. Huang and K. T. Cheng, "Error tracer: design error diagnosis based on fault simulation techniques", in IEEE transactions on CAD, 18(9): 1341-1352, September 1999.

[10] D. W. Hoffmann and T. Kropf, "Efficient Design error correction of digital circuits", in Intl. Conference on Computer Design, 2000, pp. 465-472.

[11] V. Boppana, I. Ghosh, R. Mukherjee, J. Jain and M. Fujita, "Hierarchical error diagnosis targeting RTL circuit", in Intl. Conference on VLSI Design, 2000, pp. 436-441.

[12] Maisaa Khalil, Yves Le Traon, and Chantal Robach, "Towards an Automatic Diagnosis for High-level Validation", In Proceeding Intl. Test Conference, 1998, pp. 1010-1018.