Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis

Hanno Hantson, Urmas Repinski, Jaan Raik, Maksim Jenihhin, Raimund Ubar

Tallinn University of Technology, Estonia

Abstract — Identification of the presence of design errors, i.e. verification is a well-studied field with a range of methods developed. Yet, most of the verification cycle is consumed for debugging, which consists of localization and correction of errors. Current paper presents a method for automated debug of multiple simultaneous design errors for RTL designs. We propose a critical path tracing based error localization method, which performs statistical analysis in order to rank suspected error locations. Then, an error matching approach to correction is applied implementing mutation operations. Experiments carried out in this work analyze localizing multiple erroneous data operations and their mutation-based correction. We compare two metrics of statistical analysis and show their capabilities in localizing multiple errors.

I. Introduction

Increasing design costs are the main challenge facing the semiconductor community today. Assuring the correctness of the design contributes to a major part of the problem. However, while diagnosis and repair of design errors, i.e. debug, is more time-consuming compared to error detection, it has received far less attention, both, in terms of research works and industrial tools introduced.

Consider the case where a designer has an erroneous implementation and a set of test stimuli which produce erroneous output responses on that implementation with respect to the golden responses. Automated debug of such functional design errors consists of two steps: error diagnosis and error repair. Error diagnosis identifies the portion of the design responsible for the faulty behavior, while error repair is responsible for locally modifying the functionality of the identified portion. In error diagnosis, simulation-based and formal methods are known. Error repair is performed by either applying error matching (fault model based approach) or resynthesis (fault model free approach). While the resynthesis approach is more general, it has its disadvantages which will be discussed shortly.

Design error repair for combinational circuits has been thoroughly studied for decades. There exist, both, error matching based [1][2] and resynthesis [3] approaches. There have also been attempts to generalize the above methods for design error diagnosis of sequential circuits [3] [4]. In particular, the SAT-based diagnosis and resynthesis approach developed by Smith et al. [5] has been extended to higher abstraction levels such as register-transfer level [6][7].

However, the resynthesis approach to design error repair has several limitations. Resynthesis provides a repair which is represented as a partial truth table based on the stimuli under consideration. First, this kind of repair is not readable and cannot be easily understood and verified by the design engineer. Moreover, the resynthesized erroneous portion of the design is likely to fail when new stimuli will be added to the suite. This is due to the logic optimization freedom created by the partial truth table of the portion to be repaired.

In our previous paper [8], we proposed a method for locating design errors at the source-level of hardware description language code using backtrace of mismatched and matched outputs of the system under verification on High-Level Decision Diagram (HLDD) models. In current paper we extend the method to consider localization of multiple design errors and complement it with design-error correction based on mutation operators.

The paper is organized as follows. Sections 2-4 provide the preliminaries of HLDD modeling, simulation and diagnosis. Section 2 defines the HLDD data structure. Section 3 presents simulation on HLDDs. In Section 4, we describe the method for design-error diagnosis based on HLDD backtrace [8]. Section 5 presents an error localization example. Section 6 introduces the mutation based error correction technique developed in this work. In Section 7, we discuss applying the error localization method for multiple faults. Finally, Section 8 provides the experimental results and Section 9 concludes the paper.

II. High-level decision diagrams

In this Section we define the HLDD data structure. Consider a digital system (Z, F) as a network of subsystems or components, where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, primary inputs and primary outputs of the network. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values where $Q = X \cap Y$ is the set of state variables. D(z) denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors in some variable set $Z' \subseteq Z$. Obviously, if $Z' = \{z_1, \ldots, z_n\}$ then $D(Z') = D(z_1) \times \ldots \times D(z_n)$. Let F be the set of discrete functions: $y_k = f_k(X_k)$, where $y_k \in Y$, $f_k \in F$, and $X_k \subseteq X$ (k iterates over all elements in F).

Definition 1. High-level decision diagram representing a function $f_k : D(X_k) \rightarrow D(y_k)$ is a directed acyclic multigraph G = (V, E) with a single root node and a set of terminal nodes where:

978-1-4673-2356-7/12$26.00 ©2012 IEEE

- Each non-terminal node is labeled by some input or control variable $x \in X$. We shall denote the variable of node v by x_v.
- Each terminal node w is labeled by some function $g_w : D(X_w) \rightarrow D(y_k)$, where $X_w \subseteq X_k$.
- Each edge e = (v, u) is labeled by some constant $c_e \in D(x_v)$.
- Each two edges $e_1 = (v, u_1)$ and $e_2 = (v, u_2)$ starting from the same source node are labeled by different constants $c_{e_1} \neq c_{e_2}$.
- If the node v is labeled by x_v then the number of edges starting from this node is $|D(x_v)|$.

Remark 1. Each BDD is HLDD as well, with two terminal vertices labeled by constant functions 0 and 1, and D(x) = {0, 1} for every variable x.

III. Simulation on HLDD models

HLDD models can be used for representing digital systems. In such models, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent data operations, variables or constants. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of an HLDD are calculated by other HLDDs of the system.

Simulation on high-level decision diagrams takes place as follows. Consider a situation, where all the node variables are fixed to some value. According to these values, for each non-terminal node a certain output edge will be chosen which enters into its corresponding successor node. As mentioned above, such connections between nodes are referred to as the activated edges under the given values. Succeeding each other, activated edges form in turn activated paths. For each combination of values of all the node variables there exists always a corresponding activated path from the root node to some terminal node. Let us call this path the main activated path. The simulated value of the variable represented by the HLDD will be the value of the variable labeling the terminal node of the main activated path.

Algorithm 1 presents simulation on HLDD models. The simulation process starts in the root node v_0 (line 2 of the algorithm). The node $v_{Current}$ is iteratively replaced by its successor nodes selected according to the value of $x_{v_{Current}}$ (line 4). In order to represent feedback loops in the system, in the RTL style, the algorithm takes the previous time-step value of variable x_k labeling a node v_i if x_k represents a clocked variable in the corresponding HDL (lines 5, 6). In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_k labeling HLDD nodes the previous time step value is used if the HLDD calculating x_k is ranked after current decision diagram (lines 5, 6). Otherwise, the present time step value will be used (line 8). Simulation ends when a terminal node is reached and the variable y corresponding to the simulated HLDD G_y is assigned the value $x_{v_{Current}}$ (line 12).

Algorithm 1. HLDD simulation

```
1:    SimulateHLDD(Gy)
2:      vCurrent = v0
3:      While vCurrent ∉ V^T
4:        xk = x_vCurrent
5:        If xk is clocked or
            its HLDD is ranked after G then
6:          Value = previous time-step
                value of xk
7:        Else
8:          Value = present time-step
                value of xk
9:        End if
10:       vCurrent = vCurrent^Value
11:     End while
12:     Assign y = x_vCurrent
13:   End SimulateHLDD
```

IV. HLDD backtrace for error localization

In this subsection we first present the algorithm for diagnostic tree generation using backtrace on HLDD models. Then, two analysis steps are introduced to perform error location on the set of diagnostic trees generated.

Algorithm 2 presents the recursive diagnostic tree generation on HLDDs. The process starts from the primary outputs (Line 2) and from each clock-cycle (Line 3). Subsequently, the diagnostic tree is recursively generated using the function RecursiveTreeGeneration.

Algorithm 2. HLDD-based diagnostic tree generation

```
1:    GenerateDiagnosticTree()
2:      For each primary output GO in the model
3:        For each time-step t
4:          δ(GO, t) = ∅
5:            RecursiveTreeGeneration(GO, t, δ)
6:        End for
7:      End for
8:    End GenerateDiagnosticTree

9:    RecursiveTreeGeneration(Gy, t, δ)
10:     SimulateHLDD(Gy) // See algorithm 1!
11:     For each vi at the main activated path
12:       If variable xk = x_vi at-time step t is
                not in δ then
13:         Add xk to δ
14:         If xk is not a primary input then
15:             RecursiveTreeGeneration(Gxk, t, δ)
16:         End if
17:       End if
18:     End for
19:   End RecursiveTreeGeneration
```

Algorithm 2 generates a separate diagnostic tree $\delta(G_O, t)$ for each output diagram G_O at each clock-cycle t. The resulting diagnostic tree δ is a set of pairs (x_i, t_j) that show at which time-steps t_j the variable x_i was backtraced.

In the following, two analysis steps that could be implemented for locating the design error are presented. In order to perform the analysis, let us partition the set of all diagnostic trees $\Delta = \delta_k(G_O, t)$ into failing diagnostic

trees Δ_F and passing diagnostic trees Δ_P. A diagnostic tree is failing if $\delta_k(G_O, t)$ if the simulated value of output variable $o \in Y$ differs on the faulty design at time-step t differs from the corresponding value of the golden device. Otherwise, δ_k is called a passing sequence.

Diagnosis step 1:

For each variable x_i count the number c_{FAILED} of failing diagnostic-trees $\delta_k \in \Delta_F$, where x_i is present at least in one of the pairs (x, t) of δ_k. Select the variables x_{step1} receiving the highest score c_{FAILED} as the list of suspected faults $X_{suspected}$.

Diagnosis step 2:

Perform step 1. For each variable $x_{step1} \in X_{suspected}$ count the number of passing diagnostic-trees $\delta_l \in \Delta_P$ c_{PASSED} , where x_{step1} is present at least in one of the pairs (x, t) of δ_l. Compute the score $c_{TOTAL}=c_{FAILED} / (c_{PASSED}+c_{FAILED})$ for variables x_{step2}. Select the variables x_{step2} receiving the highest score c_{TOTAL} as the new list of suspected faults.

Step 1 is more exact as it can be easily proven that at least one of the variables x_v that is labelling a vertex v along one of the main activated paths in simulated HLDDs must be also the cause of the error. However, step 2 may be unavoidable in order to guarantee a good diagnostic resolution, especially if the number of failing sequences is one or very small.

V. ERROR LOCALIZATION EXAMPLE

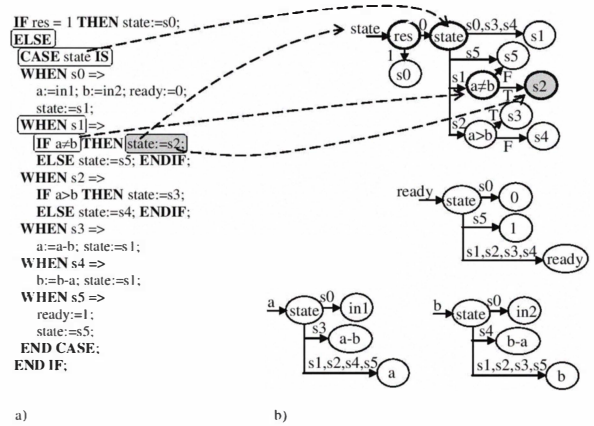

Fig. 1. a) RTL VHDL and b) its corresponding HLDD

Consider the VHDL description of the GCD design presented in Fig. 1a and its corresponding HLDD in Fig. 1b. Let there be a given set of input stimuli (e.g. a functional test) and a set of correct output responses for the stimuli obtained on a golden model. Assume that there is a design error in it such that at state s4 a faulty operation a–b is assigned to the variable b instead of the correct operation b–a. Fig. 2 shows the test sequence for the design when primary inputs in1 and in2 hold values 4 and 2, respectively. This sequence passes the test, giving a correct response that the greatest common divisor of 4 and 2 is two. In Fig. 3, another sequence is presented, which fails the test. Because of the design error, primary outputs b and ready receive erroneous values.

res	in1	in2	state	a	b	ready
1	4	2	-	-	-	-
0	-	-	s0	4	2	0
0	-	-	s1	4	2	0
0	-	-	s2	4	2	0
0	-	-	s3	4	2	0
0	-	-	s1	2	2	0
0	-	-	s5	2	2	0
0	-	-	s5	2	2	1

Fig. 2. A passing test sequence for the GCD design

res	in1	in2	state	a	b	ready
1	2	4	-	-	-	-
0	-	-	s0	2	4	0
0	-	-	s1	2	4	0
0	-	-	s2	2	4	0
0	-	-	s4	2	4	0
0	-	-	s1	2	2→-2	0
0	-	-	s5→s2	2	2→-2	0
0	-	-	s5→s3	2	2→-2	1→0

Fig. 3. A failing test sequences for the GCD design

In order to locate the design error, a diagnostic tree is generated on the HLDD model of the GCD design presented in Fig. 1b. Fig. 4 presents the diagnostic tree for the passing test shown in Fig. 2 while Fig. 5 presents the diagnostic tree for test of Fig. 3. As it can be seen from the Figures, the diagnostic tree generated by Algorithm 2 is a directed acyclic graph, where the vertices represent a subset of the time-expansion model of the design. Directed edges show relations between the variables in the simulation process. The algorithm starts at the time step an output is written (left-hand side of the figure) and continues towards the first time step (right-hand side). Borders between consecutive time steps are marked by vertical striped lines.

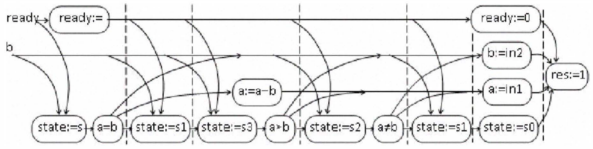

Fig. 4. Diagnostic tree for the passing test in Fig. 2

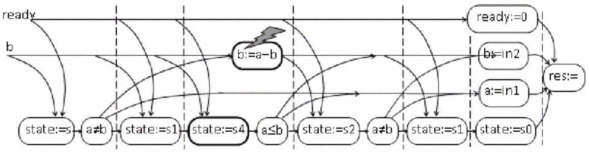

Fig. 5. Diagnostic tree for the passing test in Fig. 3

The diagnostic trees presented in Figures 4 and 5 can be used for effect-cause diagnosis of design errors. Reasoning on the diagnostic trees takes place as follows. The diagnosis tree in Fig. 4 of the passing test sequence in Fig 2 contains vertices which are unlikely to be related to the cause of the error because the sequence resulted in no mismatched outputs. However, the diagnostic tree in Fig. 5 caused two mismatched outputs ready and b because of the design error b:=a–b at the state S4. The fault should be backtraced in the tree starting from these outputs.

Indeed, the nodes labeled by b:=a–b and state:=S4 (marked by bold circles) are selected as the faults

suspected for causing the design error by the diagnosis step 2 presented in previous subsection because only these two nodes are present in the diagnostic tree of the failing sequence but are missing from the passing sequence. Thus, in this simple example they receive the highest score. In the real case there would be many failing and passing test sequences as well as there may be multiple faults. Furthermore, it may happen in most cases that it is not possible to partition the test set into sequences. Algorithm 2 takes the latter assumption. Therefore in experiments reported in current method we start backtrace at each clock cycle for each output.

The HLDD-based diagnosis is related to known debugging techniques such as program slicing [9] and critical path tracing [10]. Modeling discrete systems by a system of HLDDs may be regarded as a form of program slicing, because a separate diagram is generated for each variable x in the program, reflecting the control flow branches where assignments are made to x and including the data assigned to x. Activating paths in HLDD diagrams using Algorithm 1 is equivalent to critical path tracing. The technique of critical path tracing consists of simulating the fault-free system (true-value simulation) and using the computed signal values for backtracking all sensitized paths from primary outputs towards primary inputs in order to determine the faults that would affect the primary output. In HLDDs the same task is solved in a single run as a byproduct of simulation.

VI. MUTATION-BASED DESIGN ERROR REPAIR

Mutation analysis is a technique that was initially introduced to fulfill the task of evaluating the ability of testbenches to detect bugs in software programs. In this paper we consider applying mutation operators for repairing a faulty circuit. Subsequent to the fault localisation step described in Sections 4 and 5 mutation operators are applied to perturb the HLDD model of the RTL design in order to perform the repair. It is intuitively clear that this kind of repair may be extremely time-consuming in the worst case. The time required to repair the circuit is proportional to the product of the number of nodes, the number of mutants to be injected to each node and the number of test patterns in the test.

The design error location technique presented in previous sections allows minimizing the number of nodes where the faults have to be injected. However, it is crucial to keep the number of mutants as small as possible. In this work, the five key operators proposed in [11] have been implemented. In experiments, those five operators have provided almost the same coverage as non-selective mutation, with cost reductions of at least four times with small programs, and up to 50 times with larger programs [11]. The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, SOR, which replaces shift operators with other shift operators, and UOI, which inserts unary operators in front of expressions.

We have implemented the five operators with the following constraints and specifics. UOI currently replaces only unary operators with other unary operators. Note also that in HLDD there are no signed/unsigned variables, but signed and unsigned relational operators exist. Therefore ROR replaces, both, signed and unsigned relational operators. In AOR we also allow mutation by division and mod operations and we have included a check for the case of divide-by-zero. The reduced-5-key-operator strategy represents a do fewer strategy. The purpose would be to reduce the mutation analysis cost as much as possible.

Fig. 6 illustrates the HLDD graph perturbations for implementing the five key mutation operators on a sample diagram G_{y_out}. In HLDD models, the perturbation means simply replacement of an operator, variable or constant labeling the HLDD node by another operator, variable or constant.

Table 1 shows the list of replacements for each mutation operator. In every case the operator is substituted by another operator from the group. This is done until all operators are covered or the program is confirmed correct by simulation.

TABLE I. THE LIST OF KEY MUTATION OPERATORS

Mutation operator	List of replacements
LCR (logical connector replacement)	&, NAND, |, NOR, XOR
AOR (arithmetic operator replacement)	+, –, ·, /, MOD
UOI (unary operation insertion)	NEGATE, INVERT
SOR (shift operator replacement)	SHIFT LEFT, SHIFT RIGHT, UNSIGNED SHIFT RIGHT
ROR (relational operation replacement)	=, ≠, <, >, ≤, ≥

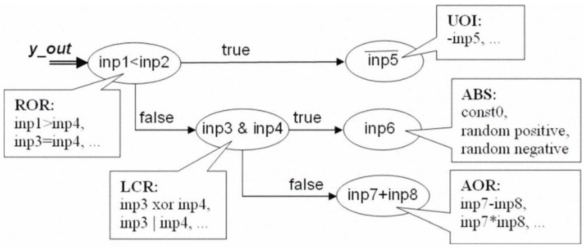

Fig. 6. "Key" mutation operators as HLDD perturbations

Algorithm 3 presents the mutation-based repair algorithm on HLDD representations. In this algorithm, first Algorithm 2 is executed in order to rank the circuit nodes according to the suspiciousness score. Then, the nodes are substituted iteratively by mutants from the same group of functions as the function labeling the node (See the groups of functions in Table 1). This iteration stops when the simulation result confirms that the repair provides output responses equal to the golden output responses.

Algorithm 3. Mutation-based repair as an HLDD perturbation

1: MutationBasedRepair()
2: Rank the nodes $v \in V$ by executing GenerateDiagnosticTree() /* Algorithm 2*/
3: For each node v
4: For each substitution p where $x_v \neq p$ in the mutation operator
5: Substitute x_v by p
6: SimulateHLDD() /* Algorithm 1*/
7: If output responses match the golden ones then
8: *Return repair "x_v to be substituted by p"*
9: End if
10: End for
11: End for
12: *Return "The design cannot be repaired by mutation"*
13: End MutationBasedRepair

VII. Localizing Multiple Design Errors

Though the statistical design error localization method presented in Section 4 is supporting localization of multiple design errors, works applying it to multiple errors have been missing. In order to evaluate when such an application would be feasible, experiments with randomly injected double design errors were carried out. The goal of the experiments was to estimate the effort needed for mutation-based repair to fix the double errors.

We propose two metrics in order to perform diagnosis for multiple errors. In metric A we rank the variables x_i according to the diagnosis step 1, where the suspiciousness score is equal to c_{FAILED}, as the primary ranking and diagnosis step 2, where the suspiciousness score is equal to $c_{TOTAL}=c_{FAILED} / (c_{PASSED}+c_{FAILED})$, as the secondary ranking, respectively. In metric B x_i are primarily ranked according to the diagnosis step 2 and when the ranks are equal secondarily ranked by diagnosis step 1.

In the following Section, experiments comparing the diagnosis capabilities of the two metrics are presented on a selection of different designs.

VIII. Experimental Results

Table 2 presents the main characteristics of the benchmarks used in the experiments and their respective test sets. The benchmarks include the Greatest Common Divisor (gcd) and the Differential Equation (diffeq) examples from the HLSynth92 and HLSynth95 academic benchmarks suite, respectively. risc is a processor example from a FUTEG research project. In addition, a commercial design core for circular redundancy check (crc) was selected [12]. The test stimuli for the academic benchmarks were generated by a hierarchical test pattern generator Decider [13] while for crc the provided functional test bench was applied. The second column reports the system complexity in terms of the number of HLDD nodes. The third column represents the number of functions in the design. Finally, the fourth column shows the number of stimuli in the test suite.

TABLE II. Benchmarks and their test sets

Design	# nodes	# fun.	# test stimuli
gcd	25	4	4000
diffeq	39	9	16855
risc	61	16	4000
crc	232	74	193

TABLE III. Design error localization experiments

Design	success rate, ratio of correct localizations		average resolution, # suspects		worst resolution, # suspects		Processing time, s
	step1	step2	step1	step2	step1	step2	
gcd	**4/4**	**4/4**	2.25	**1.00**	3	1	18.0
diffeq	**9/9**	**9/9**	3.33	**1.88**	6	3	700.0
risc	**16/16**	13/16	8.18	**1.93**	11	5	0.3
crc	**24/24**	19/24	31.83	**9.04**	50	20	0.5

In Table 3, the design error localization experiments are provided. We injected faults into the design by randomly mutating a function one-by-one, so that during each diagnosis run only one function was mutated using AOR mutation, i.e. arithmetic operator replacement (see Section 6). The column 'success rate' shows the ratio of the times the actual location of the mutation achieved **the highest rank** in relation to all diagnosis runs. The column 'average resolution, # suspects' reports the average number of suspects that received the highest score. Here, the diagnostic resolution is very good for step 2 and two or more times worse for step 1. The same trend applies to the worst resolution, which reports the worst case suspected fault list size over all the faults injected. The final column reports the run times achieved on a PC, Dual-Core CPU, 2.6GHz, 3.25GB RAM, Windows XP operating system are provided. This time includes both performing step 1 and step 2 of the diagnosis algorithm. As it can be seen, the run times are very different. They do not only depend on the circuit size but also the number of vectors and the sequential depth of the designs. The run time for step 1 is actually very much shorter than the time for steps 1 and 2 combined, because in step 1, only mismatched outputs have to be backtraced.

As shown in the previous Table 3, a majority of the errors injected in the experiments were identified as top suspects by the proposed diagnosis algorithm. Because of this location accuracy the mutation-based repair requires very small number of iterations and thus a short run-time. See table 4 which lists the average time to repair a design by applying mutation. The last column of Table 4 shows the average number of substitution functions (mutants) generated until the design was repaired.

TABLE IV. Mutation based repair experiments

Design	Average repair time, s	Average number of substitutions
gcd	0.0040	2.0
diffeq	0.0410	3.62
risc	0.0276	2.25
crc	0.0422	4.13

In Table 5, we analyzed the localization capabilities of the two metrics proposed in Section 7 for double randomly injected design errors. Again, the errors were injected using mutation operator AOR. In these experiments the sum-of-squares (sosq) design was tested in addition.

As it can be seen the accuracy of localizing double design errors for Metric B is comparable to that of single localization, sometimes even surpassing that. This can be explained by the fact that secondary ranking criterion was used to refine the localization.

TABLE V. MUTATION BASED REPAIR OF DOUBLE DESIGN ERRORS

Design	Average number of substitutions. 1 error	Average number of substitutions. 2 errors	
		Metric A	Metric B
gcd	2.0	3.5	2.5
diffeq	3.62	2.69	2.81
risc	2.25	11.67	2.5
sosq	N/A	2.75	2.75

IX. CONCLUSION

The paper presented a method for automated debug of multiple simultaneous design errors for RTL circuits. We implemented a critical path tracing based error localization method, which performs statistical analysis in order to rank suspected error locations. Then, an error matching approach to correction was applied implementing mutation operations. Experiments carried out in this work analyzed localization of multiple erroneous data operations and their mutation-based correction. We compared two metrics of statistical analysis and showed their capabilities in localizing multiple errors.

As a result of the experiments it was discovered that the localization of two simultaneous errors by one of the proposed metrics (metric B) is accurate and comparable to that of a single error localization. In some cases, the multiple error localization was even more accurate than in the case of single errors, which can be explained by the fact that secondary ranking criterion was used to refine the localization. Average repair times using mutation was just in fractions of seconds. Therefore statistical error diagnosis combined with mutation based repair appears to be a feasible approach to automated debug of multiple design errors.

ACKNOWLEDGMENT

The work has been supported by European Commission project FP7-ICT-2009-4-248613 DIAMOND, by Estonian Doctoral School in Information and Communication Technology, by Estonian Academy of Security Sciences, by European Union through the European Regional Development Fund, by Estonian Science Foundation grants 9429 and 8478.

REFERENCES

[1] J. C. Madre, O. Coudert, J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", Proceedings of ICCAD, November 1989, pp. 30-33.

[2] M. S. Abadir, J. Ferguson, T. E. Kirkland, "Logic design verification via test generation," IEEE Transactions on Computer-Aided Design, Vol. 7, No. 1, Jan. 1988.

[3] M. F. Ali, S. Safarpour, A. Veneris, M. S. Abadir, R. Drechsler, "Post-verification debugging of hierarchical designs", Proceedings of ICCAD, pp. 871-876. 2005.

[4] A. Wahba, D. Borrione, "Design error diagnosis in sequential circuits", Lecture Notes In Computer Science, Vol. 987, pp. 171 – 188, Springer, 1995.

[5] A. Smith, A. Veneris, A. Viglas, "Design Diagnosis Using Boolean Satisfiability", Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC), 2004, pp. 218-223.

[6] K.-H. Chang, et al., "Automatic Error Diagnosis and Correction for RTL Designs", Proceedings of High-Level Design and Validation Workshop (HLDVT), Nov. 2007.

[7] K.-H. Chang, I. L. Markov, V. Bertacco, "Fixing Design Errors With Counterexamples and Resynthesis," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions, vol.27, no.1, pp.184-188, Jan. 2008

[8] J. Raik, U. Repinski, R. Ubar, M. Jenihhin, A. Chepurov, "High-level design error diagnosis using backtrace on decision diagrams", 28th Norchip Conference, IEEE, 2010.

[9] M. Weiser, "Program slicing", Proceedings of the 5th International Conference on Software Engineering, pp. 439–449, IEEE Computer Society Press, March 1981.

[10] M. Abramovici, P. R. Menon, D. T. Miller, "Critical path tracing - an alternative to fault simulation", Proceedings of the 20th Design Automation Conference, pp. 214-220, June 1983.

[11] A. J. Offutt, G. Rothermel, C. Zapf, "An experimental evaluation of selective mutation", Proceedings of the Fifteenth International Conference on Software Engineering, pp. 100-107, IEEE, May 1993.

[12] EU FP6 IST STREP VERTIGO project benchmarks. URL: http://www.vertigo-project.eu/

[13] J. Raik, R. Ubar, „Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations", JETTA, Kluwer Academic Publishers, Vol. 16, No. 3, pp. 213-226, June, 2000.