IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Incremental SAT-based Accurate Auto-correction of Sequential Circuits through Automatic Test Pattern Generation

Journal:	*Transactions on Computer-Aided Design of Integrated Circuits and Systems*
Manuscript ID	TCAD-2017-0098.R1
Manuscript Type:	Full Paper-Emerging Technologies and Applications
Date Submitted by the Author:	23-Aug-2017
Complete List of Authors:	Alizadeh, Bijan Sharafinejad, Reza
Keywords:	formal verification, auto-correction, satisfiability, test generation

Incremental SAT-based Accurate Auto-correction of Sequential Circuits through Automatic Test Pattern Generation

Bijan Alizadeh, *Senior Member, IEEE*, and Seyyed Reza Sharafinejad

***Abstract*— As the complexity of digital designs continuously increases, existing methods to ensure their correctness are facing more serious challenges. Although many studies have been provided to enhance the efficiency of debugging methods, they are still suffering from the lack of scalable automatic correction mechanisms. In this paper, we propose a method for correcting multiple design bugs in gate level circuits. To reduce the correction time, an incremental satisfiability based mechanism is proposed which not only does not require a complete set of test patterns to produce a gate level implementation which does not exhibit erroneous behavior, but also will not re-introduce old bugs after fixing new bugs. The results show that our method can quickly and accurately suggest corrected gates even for large industrial circuits with many bugs. Average improvements in terms of the runtime and memory usage in comparison with existing methods are 2.8× and 6.5×, respectively. Also, the results show that our method compared to the state-of-the-art methods needs 2.6× less test patterns.**

***Index Terms*—Auto-correction, Debug, Satisfiability, Test pattern generation.**

I. Introduction

Continued growth of the complexity of digital integrated circuits as well as reduced time-to-market budget have made debugging with auto-correction mechanism in a digital design flow necessary because more than 60% of the verification effort is spent on debugging [1]. Although many researches have been done on verification and debugging of digital designs at different abstraction levels, fixing bugs (design correction) remains a manual process which is time-consuming, costly, frustrating and therefore increases time-to-market [2]. Hence, proposing efficient automatic error correction mechanism would be promising to improve overall design efficacy.

There are numerous works on debugging at different abstraction levels including gate level [3-13] and Register Transfer Level (RTL) [14-20]. The authors of [3] have mapped the problem of bug localization to a SAT problem. They assume a buggy gate level circuit as well as test patterns with correct output responses are given. The gate level circuit after inserting 2-to-1 multiplexers into their lines are converted to Conjunctive Normal Form (CNF) formula. Then, given test patterns and related correct responses as input and output constraints, respectively, are added to the previous CNF. Finally, the CNF is passed to a SAT-solver in order to find the satisfying assignments which correspond to potential bug locations. Obtaining a gate level circuit that does not exhibit erroneous behavior needs to generate test patterns exhaustively so that all parts of the circuit can be exercised which would have the scalability problem. The authors of [13] have tried to solve the same problem using a two-level Quantified Boolean Formula (QBF) formula solved by repeatedly applying normal SAT-solver. The authors of [4] have extracted a set of unsatisfiable cores from the CNF problem to accelerate debugging problem especially when multiple bugs exist. In [5], an efficient framework is presented to abstract the design by removing some state elements from the original design and those combinational logics that are only in the transitive fan-in of the abstracted state elements. Since this representation contains less logic than the original one, the size of the problem may be reduced considerably in favor of debugging. Debugging an abstract model may return abstracted state elements as error sources. In these cases, a refinement procedure is applied which replaces some of the abstracted variables with the original state elements.

The authors of [6] have proposed an approach which makes use of partial maximum satisfiability (Partial MaxSAT) to debug gate level circuits. Their formulation allows to identify all error locations both spatially and temporally. To do so, first, an approximate MaxSAT is used to find coarse solutions and then an exact SAT debugger is used to obtain final results. The authors of [7] have suggested a Partial MaxSAT based technique for automatic debugging and correcting the clock gating circuits which includes three steps: 1) erroneous node identification, 2) debugging the clock gating part using the Partial MaxSAT where the combinational and clock gating parts are taken into account as hard and soft clauses, respectively and 3) the clock gating node rectification using ranking all potential fixes based on their power saving level. In [8], an abstraction and refinement algorithm is introduced which mitigates the problems of debugging for the sequential designs. In order to decrease the complexity and error trace size, initially this algorithm divides the error trace into multiple non-overlapping time windows. Then, it iteratively analyzes each

The authors are with Design, Verification and Debugging of Embedded Systems (DVDES) Lab (https://dvdes.ut.ac.ir), School of Electrical and Computer Engineering, College of Engineering, University of Tehran, North Kargar Ave., Tehran 14395-515, Iran (e-mail: b.alizadeh@ut.ac.ir; r.sharafi@ut.ac.ir).

time windows from last time windows to the first. In each time window, it models the current and subsequent time windows using the enhanced concrete model and the path directed abstraction, respectively. To ensure that the spurious solutions are not found a refinement step is added for strengthening the abstraction.

There has been a few work on correction of digital circuit designs [9-12]. The authors of [9] have proposed a correction technique at gate level in which, first of all, a list of candidate bug locations is found by traditional SAT-based debugging techniques. Then, in order to find corrected gates, these bug locations and their related gates are passed to a mutation-based correction mechanism to exhaustively change them with other primitive gates. Afterwards, the equivalence checking is performed between the modified design and the complete set of test patterns as golden model to ensure all of the bugs are properly fixed. Note that correction time increases exponentially when the number of bugs increases due to exhaustive checking of all combinations of primitive gates. To alleviate this issue, authors of [10] have proposed our in-circuit mutation technique. In this technique we add multiplexers in candidate bug locations so that the inputs of these multiplexers are connected to all primitive gates such as AND, OR and so on. Then, the circuit including multiplexers is converted to CNF and augmented by the complete set of test patterns as golden model. Solving this problem using SAT-solver gives the corrected gates.

The authors of [11] have presented a method for debugging and rectification of combinational circuits by replacing buggy gates with look up tables (LUT). This method solves a two level QBF problem by repeatedly applying normal SAT-solver to configure LUTs so that the functionality of the circuit can be corrected. In order to convert QBF formula to a SAT formula they randomly initialize those variables that are universally quantified in QBF formula. This may increase the run time dramatically. The authors of [12] have tried to model sequential circuits using a QBF encoding. In this method, test patterns and their expected outputs are added to this model by inserting two multiplexers which help constrain the primary inputs and outputs at each time-frame. Finally, the design debugging construction is encoded to CNF formula and passed to a SAT-solver to find satisfying assignments that correspond to potential bug locations. In order to fix buggy gates, bug locations are replaced by LUTs and the correction problem is converted to a two level QBF formula. However, as it will be discussed experimentally in Section IV, solving it by the state-of-the-art QBF solvers is still a challenge especially for large circuits with a large number of LUTs, due to supporting both existential and universal quantifiers.

In [14], a debugging approach at RTL is proposed. In this work, a word level 2-to-1 multiplexer is added to error candidate statements or functional blocks in RTL code. Then, a word level SAT-solver is used to solve the resulting formula. Hence, the complexity of the debugging problem is reduced by solving it at a higher level of abstraction than gate level. The authors of [15] have proposed a method to debug and correct RTL designs. First of all, several MUXes are inserted into the HDL code of the RTL design. Then it makes use of RTL symbolic simulation, synthesis and circuit unrolling to obtain the CNF of the enriched circuit which is passed to the pseudo Boolean solver to detect potential bug locations. Finally in the correction phase, in order to fix bugs the circuit needs to be simulated which is a time consuming process. In [16], we have proposed a formal approach based on a word-level decision diagram called HED [25] to debug polynomial datapath designs. Although this method can find and correct bugs simultaneously, due to exhaustively checking all possible changes (mutations) in a design the solution space grows exponentially when the number of errors increases. A similar approach is used by the authors of [20] at RTL. The authors of [17] have improved its efficiency by reducing the debugging time. This method makes use of simple heuristic approaches to avoid computing all mutants. The method in [18] increases the scalability of the work in [16][17] without proposing a correction approach. Finally, the authors of [19] have proposed a formal debugging approach with auto-correction capability by using static slicing and dynamic ranking to increase the scalability of debugging and correction phases up to ten design bugs.

The rest of this paper is organized as follows. In Section II we describe our main contributions through a simple example. In Section III, our proposed incremental SAT-based correction method is explained in details through an example. Experimental results and a brief conclusion are given in Sections IV and V, respectively.

II. MAIN CONTRIBUTIONS

In order to discuss our main contributions in this work let us consider the problem of fixing gate level bugs in which a buggy gate level circuit as well as initial test patterns with related correct responses are given. Although existing correction methods are able to correct the circuit for given test patterns, there might be another test pattern for which the circuit behaves incorrectly. In other words, such correction methods may re-introduce old bugs after fixing new bugs due to incomplete set of test patterns and therefore there is no guarantee that the circuit works correctly for any other test patterns which are not taken into account in the process of correction.

Let us consider a gate level circuit and its erroneous version (AND gate replaced by NAND gate) as shown in Fig. 1(a) and Fig. 1(b), respectively. Suppose just one test pattern $< x_0 = 1, x_1 = 1, x_2 = 1 \,/\, y_0 = 0 >$ as initial test patterns is given that shows the circuit in Fig. 1(b) is buggy. Based on just one test vector, one solution given by existing correction methods to fix the bug is to replace NAND gate with XNOR one. Although this solution makes the circuit correct under $< 111/0 >$, there is another test vector, e.g. $< 001/1 >$, that shows the modified circuit is still buggy. In order to resolve this problem, our correction method, after fixing the bug based on given test vector, tries to find another test pattern (if possible) which shows that the circuit does not exhibit erroneous behavior. For doing so, our correction method suggests two solutions, e.g. 1) replacing NAND gate with XNOR one and 2) replacing NAND

gate with AND one as shown in Fig. 1(c). Then we check to see whether there is a test pattern under which two new gate level circuits behave differently or not. To do so, we just need to miter two solutions (i.e., XOR the outputs of the circuits) as shown in Fig. 1(d). As you can see in this figure, new test pattern $< 001/1 >$ will be generated which shows that the correct circuit is obtained by replacing NAND gate with AND not XNOR one. Note that after repeating this process until no new solution is generated, we not only could actually fix the bugs but also generate more compact test patterns which completely verify the functionality of the circuit.

Hence, our main contributions in this work are as follows:

- Proposing an automatic gate level correction method based on a new incremental SAT formula that does not need a gate-level reference specification which is not always available.
- Guaranteeing the circuit works correctly even for those test patterns that are not considered through the correction process. In other words, the accuracy of the proposed method is independent of the number of given test patterns. Moreover, it does not re-introduce old bugs after fixing new ones.
- Generating more compact test patterns to check the circuit completely without checking all combinations of the input values. Moreover, our method converges very fast to the real solution by detecting and then removing spurious solutions.

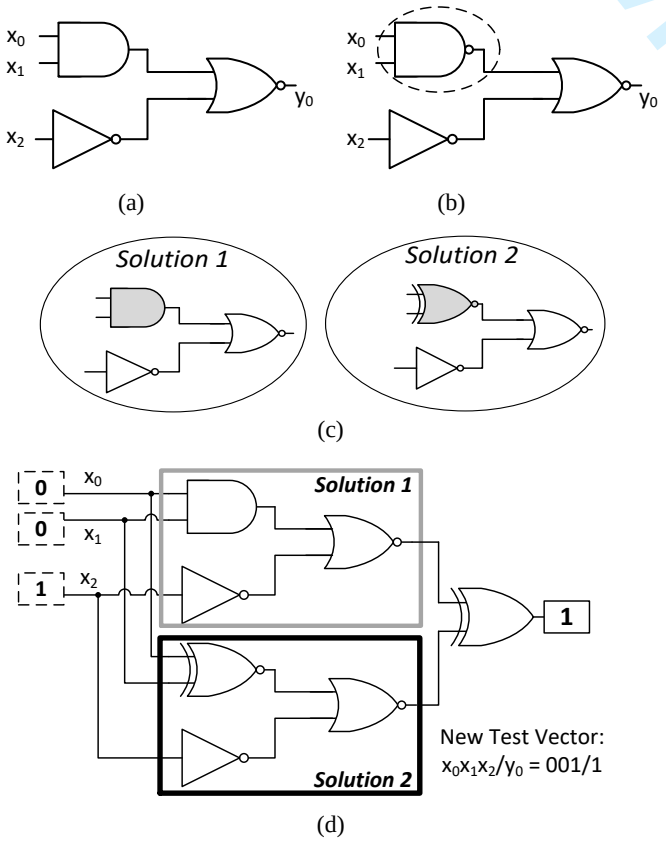


Fig. 1. A motivating example: (a) correct design, (b) erroneous design, (c) two candidate solutions, and (d) miter of two solutions to generate a new test pattern that shows under what conditions they are different.

III. Incremental SAT-based Correction Method

In this section, we explain our correction method for gate level circuits with multiple bugs. The proposed correction method gets as inputs an initial set of test patterns ($iTstP$), a buggy gate level implementation ($IMPb$) and potential bug locations ($BUGloc$). Note that, to obtain $BUGloc$, although any existing bug localization technique can be used, we have employed SAT-based debugging technique proposed in [3]. Another point is that although we have no assumption on the number initial test patterns, having more test patterns causes the method to converge very quickly to the final solution due to preventing spurious solutions. The outputs of the proposed method would be corrected gate level circuit ($IMPc$) as well as a complete set of test patterns ($cTstP$) which is initially set to $iTstP$. We say that the implementation is correct if and only if: 1) for each test pattern $tv \in iTstP$, the output of $IMPb$ equals to the expected response of tv, and 2) there is no other test pattern which indicates that the implementation behaves incorrectly. Otherwise, we say that the implementation is buggy. Although we can easily make use of a SAT-solver to check the first condition, the second condition needs to use a SAT-solver several times which makes the process complicated. This is one of our contributions in this work. Another interesting point to be noted here is the fact that, in our method, the number of initial test patterns (i.e., $|iTstP|$) has no effect on the accuracy of the result because the set of test patterns is updated during the correction process.

Note that our proposed method can be used for both combinational and sequential circuits. In order to simplify the representation of sequential circuits (with fault-free memory elements (D-FF)), we utilize *iterative logic array* (ILA) model, also called *time-frame expansion*, in which a sequential circuit is unrolled in time [9][19]. This is performed using identical copies of its combinational circuitry at different simulation cycles, where the inputs of D-FFs from cycle *i* are connected to the appropriate gates at cycle *i+1*. So, without loss of generality, in the rest of this section we assume a buggy combinational circuit needs to be corrected.

The proposed correction method consists of three main phases: 1) enriching buggy implementation, 2) finding new solutions and 3) generating test patterns which are explained in the following subsections.

A. Enriching Buggy Implementation

Although our method is extendable to the primitive gate types with more than two inputs, to keep the proposed method simple enough, we consider only the circuits with the primitive gate types AND, NAND, OR, NOR, XOR, XNOR, NOT and BUFF. In this phase, the buggy circuit is enriched by inserting 6-to-1 multiplexers [10] (shown in Fig. 2) on each line of $l \in BUGloc$ with select lines $(s_0 s_1 s_2)^l$. The basic idea behind such a modification is the fact that replacing a buggy gate on line l with other related gate (one of inputs of 6-to-1 MUX in Fig. 2) through adjusting $(s_0 s_1 s_2)^l$ appropriately, hopefully corrects the circuit. Note that, in contrast to LUT-based correction [11], we take advantage of replacing one gate at a time to correct

multiple bugs caused by more than two gates that are not close to each other. At the end of this phase, enriched implementation ($IMPe$) is obtained which is different from the original implementation ($IMPb$) as described in (1). Note that, in cases of NOT and BUFF, inserting a 2-to-1 multiplexer is sufficient.

$$Line\ l\ in\ IMPe = \begin{cases} Inserted\ MUX\ on\ line\ l\ of\ IMPb; & l \in BUGloc \\ Line\ l\ in\ IMPb; & otherwise \end{cases} \quad (1)$$

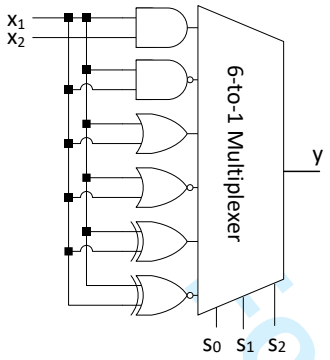


Fig. 2. An example of 6-to-1 multiplexer with 2-input primitive gates.

B. How to Find Two Solutions

ALGORITHM1 shows *FindTwoSol()* function which gets as input enriched implementation ($IMPe$), partial test patterns ($pTstP$) and generates as output two solutions ($Sol1$ and $Sol2$). For doing so, first of all, the enriched circuit is duplicated for each test pattern in $pTstP$ (lines 1-3) and the CNF of the final circuit is generated ($CNFimpe$ in line 4) as equated by (2). Note that, during duplicating the circuit, the select signals related to a MUX inserted on a line l, i.e., $(s_0s_1s_2)^l$, would be the same. This is also true during unrolling a sequential circuit.

$$CNFimpe = \prod_{i=1}^{|pTstP|}(IMPe_i(s_0s_1s_2)) \quad (2)$$

In order to generate the final CNF, constraint clauses that represent the input/output behavior of the test patterns (a set of unit clauses for the primary inputs and outputs) need to be added to the previous CNF, i.e., $CNFimpe$ (lines 5-7), as equated by (3).

$$CNFall(s_0s_1s_2) = \prod_{i=1}^{|pTstP|}(pTstP_i) \wedge (IMPe_i(s_0s_1s_2)) \quad (3)$$

In line 8 of ALGORITHM1, the final CNF formula is passed to a SAT-solver to solve the problem in (4).

$$\exists\ (s_0s_1s_2)^l\ CNFall(s_0s_1s_2)^l; l = 1\ to\ |BUGloc| \quad (4)$$

In this formula, l varies from 1 to $|BUGloc|$ in order to correct multiple bugs which is automatically formulated in CNF using the concept of cardinality constraint explained in [3]. By solving the formula in (4), we would be able to figure out how to correct the circuit. For doing so, we just need to look for those MUXes which have selected another gate rather than the original one. In other words, the problem of correction is stated as follows: *Does there exist a set of l modified gates (select lines of MUXes:$(s_0s_1s_2)^l$) such that for each input test vector, the correct primary output behavior of the circuit (added to the CNF as constraints as equated in (3)) is justified*. This statement is formally expressed by (4). If a satisfying assignment is found (line 8 of ALGORITHM1), it will be considered as a possible solution ($Sol1$ in line 9). To obtain another solution ($Sol2$ in line 12), the complement of the first solution is added to the final CNF formula so that the SAT solver returns a new solution if possible (lines 10-11).

ALGORITHM1: FindTwoSol (IMPe, pTstP)

input: enriched implementation (IMPe), partial test patterns (pTstP)
output: Solutions (Sol1 and Sol2)

```
1  foreach test pattern tv ∈ pTstP
2      Duplicate IMPe except select lines s0s1s2;
3  end for
4  CNFimpe = Generate CNF formula from duplicated circuits;
5  foreach test pattern tv ∈ pTstP
6      CNFall = Add related unit clauses of tv to CNFimpe;
7  end for
8  if (SAT-solve(CNFall) == SATISFIABLE) then
9      Sol1 = satisfying assignment returned by SAT-solve(CNFall);
10     Add Complement of Sol1 to CNFall;
11     if (SAT-solve( CNFall) == SATISFIABLE) then
12         Sol2 = satisfying assignment returned by SAT-solve(CNFall);
13     else
14         Sol2 = ∅;
15     end if
16 else
17     Sol1 = ∅;
18 end if
19 return Sol1 and Sol2;
```

C. How to Generate New Test Vector

After identifying two solutions ($Sol1$ and $Sol2$), in order to distinguish spurious solution we try to figure out under what conditions these two solutions would be different. ALGORITHM2 shows *GenNewTstVec()* function where first of all, possible corrected circuits based on two solutions are determined (lines 1-2 of ALGORITHM2). Then, in line 3 of ALGORITHM2, the outputs of two circuits are connected with a miter (XOR gate). Finally, its CNF formula is generated (line 4) and passed to a SAT-solver. If a satisfying assignment is found (line 5), it will be considered as a new test pattern ($NewTstVec$ in line 6) which indicates under what conditions two gate level corrections behave differently.

ALGORITHM2: GenNewTstVec (IMPe, Sol1, Sol2)

input: enriched implementation (IMPe), first solution (Sol1), second solution (Sol2)
output: new test pattern (NewTstVec)

```
1  Cir1 = IMPe correction based on Sol1;
2  Cir2 = IMPe correction based on Sol2;
3  mCir = miter(Cir1, Cir2);
4  CNFmCir = Generate CNF formula from mCir;
5  if (SAT-solve(CNFmCir) == SATISFIABLE) then
6      NewTstVec = satisfying assignment by SAT-solve(CNFmCir);
7  else
8      NewTstVec = ∅;
9  end if
10 return NewTstVec;
```

D. Put Them All Together - Proposed Correction Method

As mentioned before, the proposed correction method gets as inputs an initial set of test patterns ($iTstP$), a buggy gate level implementation ($IMPb$) and potential bug locations ($BUGloc$) and generates the corrected gate level circuit ($IMPc$) as well as a complete set of test patterns ($cTstP$) as outputs. ALGORITHM3 shows the proposed correction method which starts by enriching the buggy implementation (line 2 of ALGORITHM3). Then the algorithm tries to correct the buggy implementation by changing one gate or more at potential bug locations. This process repeats until a solution is obtained (see repeat-until loop in lines 3-29 of ALGORITHM3). It should be noted that we assume the circuit would be correctable with the existing gates. Otherwise, our method is not able to correct the circuit due to incorrectly selection of bug locations (lines 24-26 of ALGORITHM3).

The process of correction starts by finding two solutions (*FindTwoSol()* function in line 4 of ALGORITHM3). If a second solution does not exist (line 6), we can guarantee that the first solution is the final correction (See Theorem 1 in subsection III-E). Therefore the implementation is corrected based on $Sol1$ and the algorithm finishes (lines 7-8). Otherwise, in order to show under what conditions at the primary inputs two solutions ($Sol1$ and $Sol2$) would be different, the proposed method generates a new test pattern if possible (*GenNewTstVec()* function in line 10 of ALGORITHM3). If a new test pattern is generated (line 11), it is added to the set of test patterns (line 12) and the loop is repeated. If, however, a new test pattern is not generated (line 15), although it indicates that two solutions are functionally equivalent under existing test patterns, they may be spurious solutions. To ignore current solutions the complement of $Sol1$ or $Sol2$ is added to the *CNFall* (line 16) and the process of generating two other solutions which are different from the first ones and finding new test pattern are repeated. Note that if the first solution does not exist (line 20), $OldSol$ obtained in the previous round (line 17) would be the final correction (See Theorem 2 in subsection III-E). Finally, $OldSol == \varnothing$ indicates that our method is not able to correct the circuits due to incorrect bug locations (lines 24-26).

Note that, in contrast to [11] which assumes a reference specification is available, we just ask the designer to determine the correct response of a new test vector. One of the advantages of our method is that new solutions never re-introduce old bugs that are satisfied using previous partial test patterns. Another advantage is the fact that in addition to multiple bug correction, a complete set of test patterns is generated.

E. Correctness Proof

The following theorems formally proves the correctness of the proposed method.

Theorem 1: Let $cTstP$ be the complete set of test patterns returned by ALGORITHM3. ALGORITHM3 terminates if $Sol1 \neq \varnothing$ and $Sol2 = \varnothing$ (see lines 5-6), and $Sol1$ is the final correction (see lines 7-8) which is not a spurious solution. In other words, there is no new test vector, $tv \notin cTstP$, such that $\prod_{i=1}^{|cTstP|+1}(cTstP_i) \wedge (IMPe_i(Sol1)) \equiv UNSAT$, where $cTstP_{|cTstP|+1} = tv$.

Proof: Based on (4) and $1 \neq \varnothing$, we can write:

$$\prod_{i=1}^{|cTstP|}(cTstP_i) \wedge (IMPe_i(Sol1)) \equiv SAT \quad (5)$$

Also, based on (4) and $2 = \varnothing$, we can write:

$$\forall S \neq Sol1 \mid \prod_{i=1}^{|cTstP|}(cTstP_i) \wedge (IMPe_i(S)) \equiv UNSAT \quad (6)$$

We need to prove (7) which means that there is no test vector, $tv \notin cTstP$ ($cTstP_{|cTstP|+1} = tv$), such that $Sol1$ gives a wrong result.

$$\prod_{i=1}^{|cTstP|+1}(allTstP_i) \wedge (IMPe_i(Sol1)) \equiv SAT \quad (7)$$

We use Reductio ad absurdum. Let us assume (7) is not correct which means that there exists a test vector, $tv \notin cTstP$, such that $\prod_{i=1}^{|cTstP|+1}(cTstP_i) \wedge (IMPe_i(Sol1)) \equiv UNSAT$, where $cTstP_{|cTstP|+1} = tv$. So, there should be another solution $Sol1' \neq Sol1$ such that $\prod_{i=1}^{|cTstP|+1}(allTstP_i) \wedge (IMPe_i(Sol1')) \equiv SAT$. Obviously, $Sol1'$ should satisfy the test patterns in $cTstP$. Therefore, $\prod_{i=1}^{|cTstP|}(cTstP_i) \wedge (IMPe_i(Sol1')) \equiv SAT$ which contradicts (6). This completes the proof. ■

ALGORITHM3: Incremental SAT-based Correction Method

```
   inputs: buggy implementation (IMPb), initial test patterns (iTstP),
           potential bug locations (BUGloc)
   outputs: corrected implementation (IMPc),
            complete test patterns (cTstP)
1  cTstP = iTstP;
2  IMPe = EnrichBuggyImplementation(IMPb, BUGloc);
3  repeat
4     (Sol1, Sol2) = FindTwoSol (IMPe, cTstP, C);
5     if (Sol1 ≠ ∅) then
6        if (Sol2 == ∅) then
7           IMPc = Modify IMPe based on Sol1;
8           return (IMPc, cTstP);
9        else
10          nTstv = GenNewTstVec (Sol1, Sol2);
11          if (nTstv ≠ ∅) then
12             cTstP = nTstv ∪ cTstP;
13             OldSol  = ∅;
14             Remove all constraints (added in line 16);
15          else
16             Add constraints so that Sol1/Sol2 is not
               generated again;
17             OldSol  = Sol1;
18          end if
19       end if
20    else
21       if (OldSol ≠ ∅) then
22          IMPc = Modify IMPe based on OldSol;
23          return (IMPc, cTstP);
24       else
25          Our method is not able to correct the circuit due to
            incorrect bug locations;
26          return;
27       end if
28    end if
29 until ( Sol1 == ∅ or  Sol2 == ∅)
```

Theorem 2: Let $cTstP$ be the complete set of test patterns returned by ALGORITHM3. ALGORITHM3 terminates if $Sol1 = \varnothing$ and $OldSol \neq \varnothing$ (see lines 20-21), and $OldSol$ is the final correction (see lines 22-23) which is not a spurious solution. In other words, there is no new test vector, tv, such that $\prod_{i=1}^{|cTstP|+1}(cTstP_i) \wedge (IMPe_i(OldSol)) \equiv UNSAT$, where $cTstP_{|cTstP|+1} = tv$.

Proof: The proof is similar to that of Theorem 1 just by replacing $Sol1$ and $Sol2$ by $OldSol$ and $Sol1$, respectively. ■

F. Example

In order to clarify the proposed auto-correction method, let us consider a simple sequential circuit as well as its buggy version (AND gate is replaced by a NAND gate) shown in Fig. 3(a) and Fig. 3(b), respectively. The initial value of the D-FF is set to 0. Let us consider $BUGloc = \{l_0, l_2\}$ and $iTstP = \{< x_0^1 x_1^1 x_2^1 / y^1 = 000/0\,, x_0^2 x_1^2 x_2^2 / y^2 = 000/0 >\}$ as potential bug locations and initial test patterns for two cycles, respectively. $cTstP$ is initialized by $iTstP$ and therefore $cTstP = \{< 000/0{,}000/0 >\}$. As mentioned before (line 2 of ALGORITHM3), first of all, the buggy circuit is enriched by adding 6-to-1 multiplexers in potential bug locations as shown in Fig. 4. After unrolling the enriched circuit for two cycles, two possible solutions using ALGORITHM1 are found:

- *Solution 1:* $s_0 s_1 s_2(l_0) = 100$, i.e., XOR gate in l_0, $s_0 s_1 s_2(l_2) = 100$, i.e., XOR gate in l_2.
- *Solution 2:* $s_0 s_1 s_2(l_0) = 000$, i.e., AND gate in l_0, $s_0 s_1 s_2(l_2) = 000$, i.e., AND gate in l_2.

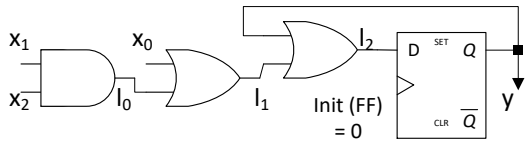
Then, the miter generates a new test pattern which shows under what conditions these two solutions would be different (line 3 of ALGORITHM2). $< x_0^1 x_1^1 x_2^1 / y^1 = 100/1\,, x_0^2 x_1^2 x_2^2 / y^2 = 000/1 >$ as a new test pattern is generated (line 10 in ALGORITHM3) and added to the complete set of test patterns, i.e., $cTstP$ (line 12 in ALGORITHM3). So $cTstP = \{< 000/0, 000/0 >, < 100/1, 000/1 >\}$. Again, we try to find two other solutions based on new set of test patterns:

- *Solution 1:* $s_0 s_1 s_2(l_0) = 100$, i.e., XOR gate in l_0, $s_0 s_1 s_2(l_2) = 010$, i.e., OR gate in l_2.
- *Solution 2:* $s_0 s_1 s_2(l_0) = 000$, i.e., AND gate in l_0, $s_0 s_1 s_2(l_2) = 100$, i.e., XOR gate in l_2.

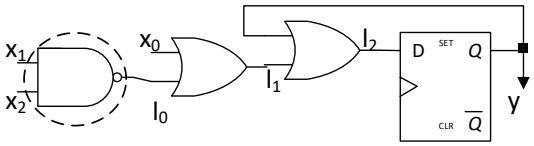
New test pattern $< 100/1, 100/1 >$ is added to $cTstP$. So $cTstP = \{< 000/0, 000/0 >, < 100/1, 000/1 >, < 100/1, 100/1 >\}$. Again, we try to find two other solutions based on new set of test patterns, $cTstP$:

- *Solution 1:* $s_0 s_1 s_2(l_0) = 000$, i.e., AND gate in l_0, $s_0 s_1 s_2(l_2) = 010$, i.e., OR gate in l_2.
- *Solution 2:* $s_0 s_1 s_2(l_0) = 100$, i.e., XOR gate in l_0, $s_0 s_1 s_2(l_2) = 100$, i.e., XOR gate in l_2.

New test pattern $< 010/0, 000/0 >$ is added to $cTstP$. So $cTstP = \{< 000/0, 000/0 >, < 100/1, 000/1 >, < 100/1, 100/1 >, < 010/0, 000/0 >\}$. After trying to find two solutions, the only solution would be $s_0 s_1 s_2(l_0) = 000$, i.e., AND gate in l_0, $s_0 s_1 s_2(l_2) = 010$, i.e., OR gate in l_2 and the second solution would be empty. It shows that the bug is fixed by replacing NAND gate with a AND gate. This way, we could make the circuit correct just by using four test patterns while other methods [9][10][12] need eight test patterns to do so.



(a)



(b)

Fig. 3. (a) A simple original sequential circuit, and (b) buggy circuit.

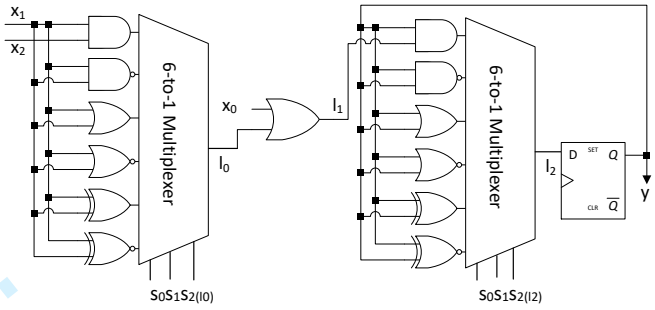


Fig. 4. Enriched circuit.

IV. Experimental Results

In order to demonstrate the effectiveness of the proposed method, we perform various experiments on ISCAS'85, ISCAS'89 and ITC'99 benchmark circuits [21]. Table 1 shows the design statistics related to the benchmark circuits taken into account in this paper. Columns 2, 3 and 4 denote the number of inputs and outputs, the number of gates and the number of D flip-flops, respectively. Note that, sequential circuits are unrolled for nine cycles (except for experiment 6) based on the ILA representation discussed before. The proposed method has been implemented in C++ and performed on a 2.4 GHz Intel core i7 Haswell CPU (having 4 processor cores) with 6GB RAM running on Windows 8.1 where MiniSAT has been used as a SAT-solver [22][23] with a timeout of 18000 seconds for each SAT problem. The erroneous circuits are created by randomly changing the type of a single or multiple gates as well as their wires [10]. In other words, the structure (topology) of the circuit as well as individual gates may be wrong.

We carry out seven experiments as will be explained in the following subsections. In the first experiment, we compare our results with those of [9][10][11][12]. In the experiments, we demonstrate the scalability of the proposed method in terms of the number of potential bug locations (each potential bug location containing a single gate), the number of bugs (each bug containing a single faulty gate), the number of initial test patterns, the number test patterns generated and the number of

time frame expansions. The final experiment shows how our method behaves when there is no solution.

TABLE 1
CIRCUIT DETAILS OF ISCAS AND ITC BENCHMARKS

Circuit Name	# Inputs & Outputs	# Gates	# D-FF
c22	7	10	0
c499	42	160	0
c1355	73	546	0
c2670	363	1193	0
c3540	72	1669	0
c5315	301	2307	0
c6288	64	2416	0
c7552	315	3512	0
s1238	32	508	18
s1488	27	653	6
b14	86	10098	245
b20	54	20226	490
b22	54	29951	735
b17	134	32326	1415
b18	60	114621	3320

A. *Experiment 1: comparison with existing methods*

In the first experiment, we compare our results with those of the state-of-the-art multiple bugs correction techniques [9][10][11][12] in terms of the CPU time and the required number of test patterns. Table 2 tabulates the run times in seconds and memory usage in megabytes for different number of gate level bugs for combinational and sequential circuits. In this table, column *#Bugs* indicates the number of bugs or LUTs (in the cases of [11] and [12]) inserted into the circuits. Note that although the authors of [12] have not provided a correction mechanism, we have formulated the correction problem as a two level QBF formula, i.e., ∃∀∃-queries, resolved using a QBF solver called RAReQS [24]. The method in [11], however, solves the same problem using a SAT-solver incrementally while it assumes that the corrected gate level circuit as a specification is available. In general, the main drawback of the methods presented in [9][10][11][12] is the fact that a complete set of test patterns or a complete specification are needed to guarantee the circuit is completely corrected. Otherwise, these methods may re-introduce old-bugs after fixing new ones.

The results in Table 2 show that our proposed method outperforms existing correction methods in terms of the correction time (and memory usage) which has been reduced by 2.5× (3.81×), 1.15× (1.12×), 1.12× (1.04×) and 6.34× (19.93×), in comparison with those of [9], [10], [11] and [12], respectively. The reason for this behavior is the fact that existing methods need a complete set of test patterns in order to guarantee that all bugs have been fixed. Replicating the circuit for each test pattern makes the CNF or QBF formula too much complicated so that the correction time increases dramatically. While our method tries to find more compact test patterns during correcting the circuit which makes the problem to be solved by the SAT-solver as simple as possible and therefore the run time decreases significantly. Another interesting point to be noted here is the fact that considering the correction problem as a two level QBF formula and trying to solve it using QBF solvers is just applicable to small circuits and it exceeds the time out limit of 18000 seconds (TO in Table 2), whereas the proposed method could solve all of cases within several minutes. Note that N.A. (Not Applicable) in Table 2 indicates that the method is not applicable due to the time out.

TABLE 2
CPU TIME AND MEMORY USAGE COMPARISON BETWEEN THE PROPOSED METHOD AND EXISTING METHODS [9][10][11][12]

Benchmark		Proposed method		QBF-based method in [12]		Incremental SAT method in [11]		Mutation-based method in [9]		In-circuit based method in [10]	
Circuit name	**#Bugs**	**Time (sec)**	**Mem (MB)**	**Time (sec)**	**Mem (MB)**	**Time (sec)**	**Mem (MB)**	**Time (sec)**	**Mem (MB)**	**Time (sec)**	**Mem (MB)**
c22	3	4	0.44	8	0.76	6	0.32	7	0.41	4	0.45
c499	4	25	0.51	432	1.92	36	0.55	78	1.09	27	0.59
c1355	5	42	0.63	865	2.36	49	0.68	119	1.39	45	0.73
c2670	7	89	0.77	11,714	51.42	154	0.82	412	2.34	101	1.02
c3540	8	256	1.03	TO	N.A.	308	1.12	1,397	3.58	310	1.32
c5315	8	569	1.19	TO	N.A.	609	1.37	3,635	4.33	681	1.45
c6288	8	617	1.24	TO	N.A.	628	1.42	3,778	5.41	713	1.78
c7552	8	643	1.56	TO	N.A.	713	1.85	3,833	6.27	723	2.13
s1238	4	37	0.64	704	3.15	49	0.79	129	2.64	48	0.90
s1488	5	125	0.15	TO	N.A.	144	1.24	428	5.84	137	1.58
b14	7	863	2.83	TO	N.A.	941	3.13	3,751	16.65	891	3.46
b20	8	2,745	6.04	TO	N.A.	3,208	6.23	7,887	25.75	3,196	6.92
b22	8	4,144	15.85	TO	N.A.	4,305	16.11	14,949	53.41	4,921	17.23
b17	8	5,3011	17.65	TO	N.A.	5,177	17.92	TO	N.A.	5,540	18.41
b18	8	15,364	62.87	TO	N.A.	TO	N.A.	TO	N.A.	TO	N.A.
Average run time and memory usage improvement of the proposed method				6.34×	19.93×	1.12×	1.04×	2.5×	3.81×	1.15×	1.12×

To compare our method with others in terms of the number of test patterns required to completely correct buggy circuits, we report the number of test patterns in Table 3. For instance, in the case of the c7552 circuit, other methods require 217 test patterns to fix all of eight bugs (*#Bugs* column) injected into the c7552 circuit. While our method can start with 37 test patterns (column *#InitTstV*) and generates 46 more test patterns (column *#PartialTstV*), i.e., totally 83 test patterns instead of 217, in order to fix all of eight bugs. As exhibited in this table, our method with a lower number of test patterns can fix all of injected bugs into the circuits. On average, up to 2.6× reduction in the required number of test patterns is achieved.

TABLE 3
NUMBER OF TEST PATTERNS REQUIRED TO COMPLETELY CORRECT BUGGY CIRCUITS

Benchmark		Proposed method		Other methods
Circuit	**#Bugs**	**#InitTstV**	**#PartialTstV**	**#CompleteTstV**
c22	3	3	3	7
c499	4	8	10	63
c1355	5	12	21	85
c2670	7	18	29	99
c3540	8	27	33	146
c5315	8	29	26	120
c6288	8	18	5	35
c7552	8	37	46	217
s1238	4	12	6	68
s1488	5	71	75	359
b14	7	389	452	3,883
b20	8	1,245	1,867	7,339
b22	8	2,565	2,852	12,928
b17	8	2,217	3,310	13,832
b18	8	8,741	9,560	48,253
Test pattern reduction compared to others				2.6×

B. Experiment 2: impact of the number of potential bug locations on the run time

In the second experiment, we show the impact of increasing the number of potential bug locations on the run time of the proposed method. Note that by increasing the number of potential bug locations, the number of correction multiplexers should be added to the enriched model increases as well. It causes the CNF formula size to grow which affects the runtime of the test pattern generation process. Fig. 5 shows the run time of the c7552, c5315, c3540 and c2670 circuits in terms of the number of potential bug locations. We have supposed that the number of bugs and the number of initial test patterns are 8 and 100, respectively. As can be seen, the run time increases linearly where its slope is mild. This means that the time complexity of the proposed method is almost independent of the number of potential bug locations and therefore it is scalable enough to correct large gate-level circuits with even more than hundred bugs.

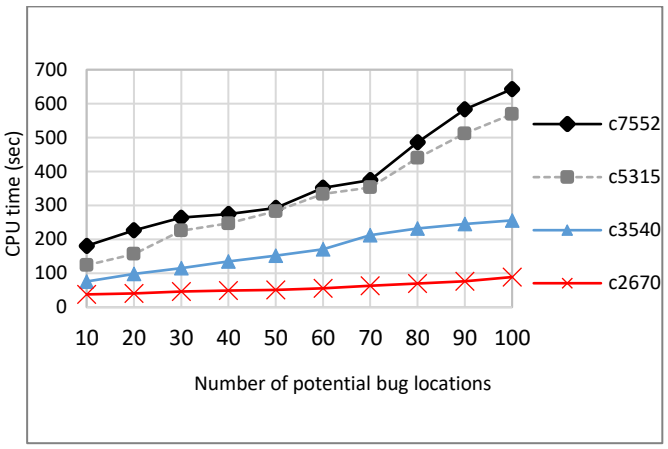


Fig. 5. Impact of the number of potential bug locations on the run time.

C. Experiment 3: effect of the number of bugs on the run time

One key point to be taken into account when an auto-correction method is proposed is again related to its time complexity but when the number of bugs increases. Fig. 6 illustrates the relationship between the number of bugs and the overall run time of the c7552, c5315, c3540 and c2670 when 50 potential bug locations are taken into account. It is obvious that the time complexity of the proposed method is linearly increasing when the number of bugs increases.

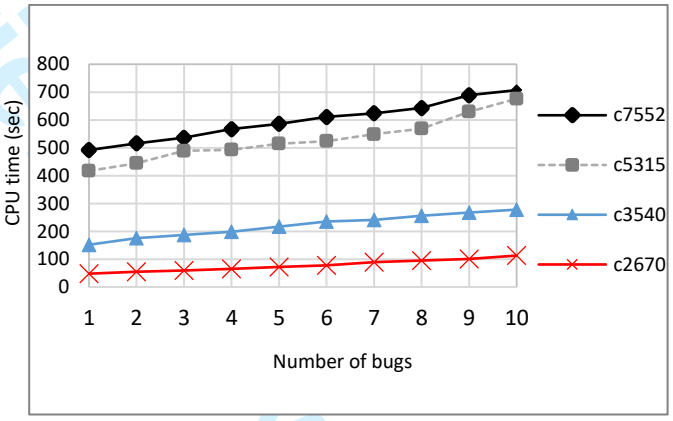
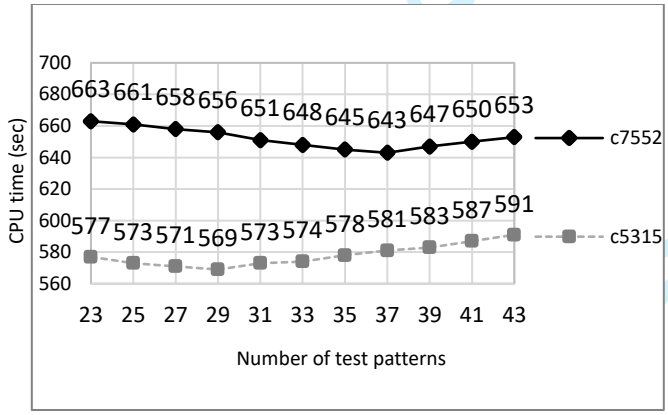


Fig. 6. Effect of the number of bugs on the run time.

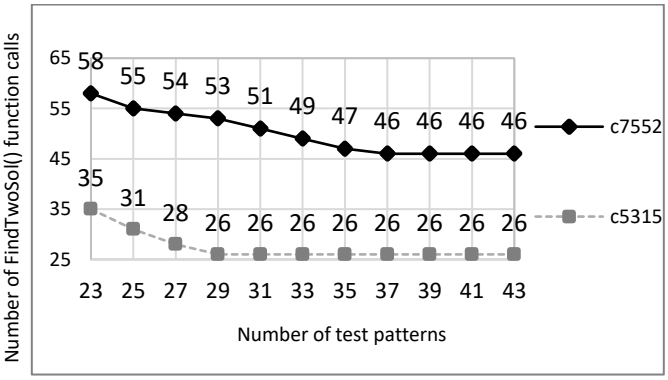
D. Experiment 4: impact of the number of initial test patterns on the run time and the number of solutions

As mentioned before, one of the advantages of our method is the fact that its accuracy is independent of the initial test patterns (*iTstP* in ALGORITHM3) given by the designer. In contrast to existing methods which are very dependent to the given test patterns to fix the bugs so that old bugs may be re-produced after fixing new ones, our method is able to start by even a few test patterns that are updated through the correction process until all bugs are fixed. Fig. 7(a) shows the relationship between the number of initial test patterns and the run time of the c5315 and c7552 circuits when 50 potential bug locations and eight bugs are considered. Fig. 7(b) illustrates the relationship between the number of initial test patterns and how many times *FindTwoSol()* function (ALGORITHM1) is called.

An interesting point here is the fact that, the curves in Fig. 7(a) have a minimum point in which the effects of the run time and the number of solutions are balanced. For example in the case of the c7552 circuit, 37 test patterns would be the optimal number of initial test patterns (the minimum point in the curve) for fixing eight injected bugs when our method just needs to generate 46 more test patterns to do so (see Fig. 7(b)). If, however, the number of test patterns is less than 37, other solutions (spurious ones) are found which do not exactly target the bugs injected into the circuits which obviously increases the run time. In other words, by increasing the number of initial test patterns, our method converges faster to the exact solution due to removing spurious solutions as early as possible. On the other hand, when the number of test patterns becomes greater than 37, we actually just spend time and memory due to duplicating the circuit for each test vector, without getting better solutions. In other words, the number of solutions is fixed when the number of test patterns exceeds 37 as you can see in Fig. 7(b). Note that, this minimum point for the c5315 circuit is 29.



(a)



(b)

Fig. 7. Impact of the number of initial test patterns on: (a) the run time, and (b) the number of solutions.

E. Experiment 5: effect of the number of generated test patterns on the run time

Fig. 8 shows the impact of the number of generated test patterns on the run time of the c5315, c7552, c3540 and c2670 circuits when 50 potential bug locations and eight bugs are considered. Note that, the optimal number of initial test patterns reported in Fig. 7 is used as the initial test patterns. To analyze the data in Fig. 8, let us consider the c7552 circuit in which 37 initial test patterns (see Fig. 7) is taken into account. As can be seen, the run time to generate the first 10 new test patterns is increased by about 40% while the run time to generate the rest of new test patterns is increased by about 60%. The main reason for this difference is that by increasing the number test patterns (a large *cTstP* in ALGORITHM3), the number of equivalent solutions which generate empty test patterns increases as well (line 15 of ALGORITHM3). It makes ALGORITHM3 (lines 3-29) to spend more time to find new solutions which generate non-empty test patterns. The results show that the proposed method is scalable enough in this regard.

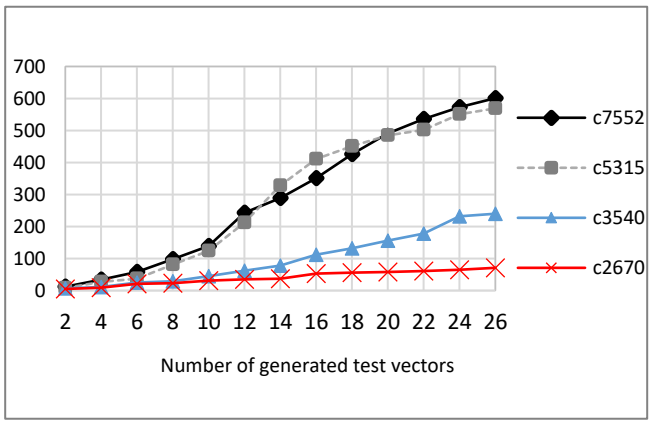


Fig. 8. Effect of the number of generated test patterns on the run time.

F. Experiment 6: effect of the number of time frame expansions on the run time

In order to show that the proposed correction method is applicable to sequential circuits as well, in this experiment, we have applied our method to three sequential circuits (b17, b22, and b20). Fig. 9 shows the effect of the number of time frame expansions (ILA model) on the run time when 50 potential bug locations and eight bugs are considered. By increasing the number of time frame expansions, although the CNF size increases, our method is able to solve the problem when the run time increases linearly due to fixed number of primary inputs and outputs in ILA model.

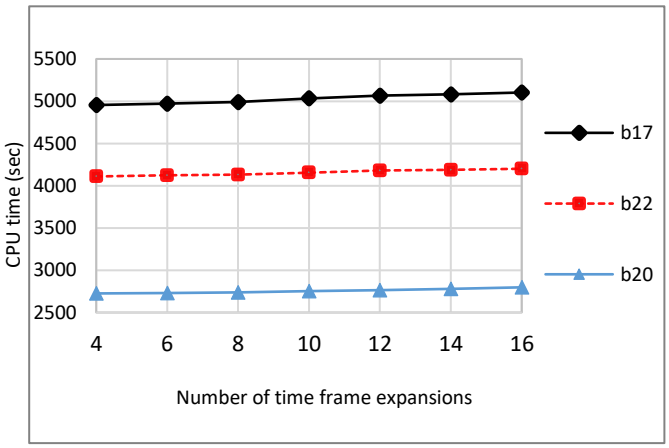


Fig. 9. Effect of the number of time frame expansions on the run time.

G. Experiment 7: when there is no solution

In the last experiment, in order to show the behavior of our method in terms of the run time when there is no solution (see lines 24-26 of ALGORITHM3), we intentionally remove some real bug locations from potential bug locations (*BUGloc*). The results in comparison with those of [11] are reported in Table 4 where *#Iteration* indicates the number of iterations that incremental SAT is repeated. The results show that our method could actually prove there is no solution 1.6× faster than that of [11] with 2.2× less iterations. As mentioned before, the method in [11] solves two level QBF formula equated in (5) by repeatedly applying normal SAT-solver to configure LUTs (by determining $\vec{v}$) so that the functionality of the circuit can be corrected.

$$\exists \vec{v}\, \forall \vec{x}\; IMPL(\vec{v},\vec{x}) = SPEC(\vec{x}) \qquad (5)$$

One of its limitations is the fact that a specification as a golden gate level circuit (see *SPEC* in (5)) is needed which is not always available. Our method, in contrast, asks the designer to specify the correct response of a new test vector. Another limitation is related to the initial values for those variables that are universally quantified, i.e., $\vec{x}$, in (5). Choosing them not suitably increases the run time dramatically.

TABLE 4
RESULTS WHEN THERE IS NO SOLUTION

Benchmark		Proposed method		Method in [11]	
Circuit	#Bugs	Time (s)	#Iteration	Time (s)	#Iteration
c499	2	5	2	10	3
c1355	2	8	3	15	8
c2670	4	13	4	21	10
c3540	4	46	6	82	12
c5315	6	67	6	117	14
c7552	6	112	8	158	16
Improvements compared to [11]				1.6×	2.2×

V. Conclusions and Future Work

In this paper, we have presented a novel scalable method for automatic correction of gate-level circuits with multiple bugs. Our method is based on incremental SAT and the basic idea is to generate two solutions, instead of one, at each iteration of the correction process which help us to generate new test pattern if possible. In other words, our method not only enables us to fix multiple errors, but also generates more compact test patterns that guarantee the circuit is completely corrected and there is no test pattern under which the modified circuit behaves incorrectly. The results show that our method significantly reduces the correction time, memory usage and the number of test patterns by 2.8×, 6.5× and 2.6×, respectively, in comparison with existing methods presented in [9][10][11][12]. In the future, we plan to make our method applicable to RTL designs and we also want to use MaxSAT-based techniques to further reduce the run time.

References

[1] H. Foster, "Assertion-Based Verification: Industry Myths to Realities (invited tutorial)," in *Proc. Of International Conference in Computer Aided Verification (CAV)*, pp. 5-10, 2008.

[2] P. Rashinkar, P. Paterson, and L. Singh, System-on-a-Chip Verication: Methodology and Techniques. Boston, MA: Kluwer, 2000.

[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault Diagnosis and Logic Debugging Using Boolean Satisfiability," in *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 10, pp. 1606-1621, 2005.

[4] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, "Using Unsatisable Cores to Debug Multiple Design Errors," in *Proc. of GLSVLSI*, pp. 77-82, 2008.

[5] S. Safarpour, and A. Veneris, "Automated Design Debugging with Abstraction and Refinement," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 10, pp. 1597-1608, 2009.

[6] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated Design Debugging with Maximum Satisfiability," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, no. 11, pp. 1804-1817, 2010.

[7] B. Le, S. Dipanjan, and A. Veneris. "Reviving Erroneous Stability-based Clock-gating using Partial Max-SAT," in *Proc. Of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 717-722, 2013.

[8] B. Keng, and A. Veneris, "Path-Directed Abstraction and Refinement for SAT-Based Design Debugging," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 32, no. 10, pp. 1609-1622, 2013.

[9] P. Behnam, B. Alizadeh, and Z. Navabi, "Automatic Correction of Certain Design Errors using Mutation Technique," in *Proc. of European Test Symposium (ETS)*, pp. 1-2, 2014.

[10] P. Behnam, and B. Alizadeh, "In-circuit Mutation-based Automatic Correction of Certain Design Errors using SAT Mechanisms," in *Proc. of Asian Test Symposium (ATS)*, pp. 199-204, 2015.

[11] S. Jo, T. Matsumoto, and M. Fujita, "SAT-based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions," *in Proc. Of Asian Test Symposium (ATS)*, pp. 19-24, 2012.

[12] H. Mangassarian, and A. Veneris, "Robust QBF Encodings for Sequential Circuits with Applications to Verification, Debug and Test," in *IEEE Transactions on Computers (TC)*, vol. 59, no. 7, pp. 981-994, 2010.

[13] H. Riener and G. Fey, "Exact Diagnosis using Boolean Satisfiability," in Proc. of *International Conference on Computer-Aided Design (ICCAD)*, pp. , 2016.

[14] S. Mirzaeian, F. Zheng, and T. Cheng, "RTL Error Diagnosis Using a Word-Level SAT-Solver," in *Proc. Of International Test Conference (ITC)*, pp. 1-8, 2008.

[15] K. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic Error Diagnosis and Correction for RTL Designs," in *Proc. Of High Level Design Verification and Test (HLDVT)*, pp. 683-688, 2010.

[16] B. Alizadeh, "A Formal Approach to Debug Polynomial Datapath Designs," in *Proc. Of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 683-688, 2012.

[17] B. Alizadeh, P. Behnam, "Formal Equivalence Verification and Debugging Techniques with Auto-Correction Mechanism for RTL Designs," in *Elsevier Microprocessors and Microsystems – Embedded Hardware Design (MICPRO)*, vol. 37, no. 8, pp. 1108–1121, 2013.

[18] S. Sadeghi-kohan, P. Behnam, B. Alizadeh, M. Fujita, and Z. Navabi, "Improving Polynomial Datapath Debugging with HEDs," in *Proc. Of European Test Symposium (ETS)*, pp. 1-6, 2014.

[19] B. Alizadeh, P. Behnam, and S. Sadeghi-Kohan, "A Scalable Formal Debugging Approach with Auto-correction Capability based on Static Slicing and Dynamic Ranking for RTL Datapath Designs," in *IEEE Transactions on Computers (TC)*, vol. 64, no. 6, pp. 1564-1578, 2015.

[20] J. Raik, U. Repinski, A. Chepurov, H. Hantson, R. Ubar, and M. Jenihhin, "Automated Design Error Debug using High-Level Decision Diagrams and Mutation Operators," in *Elsevier Journal of Microprocessors and Microsystems – Embedded Hardware Design (MICPRO)*, vol. 37, no. 4-5, pp. 505-513, 2013.

[21] ISCAS & ITC bench marks available online at: http://pld.ttu.ee/~maksim/benchmarks/

[22] "MiniSAT ver. 2.2." available online at: http://minisat.se/MiniSat.html

[23] N. Eén, and N. Sörensson, "An extensible SAT-solver," in *Proc. of International Conference on Theory and Applications of Satisfiability Testing*, pp. 502–518, 2003.

[24] M. Janota, W. Klieber, J. Marques-Silva and E. Clarke, "Solving QBF with Counterexample Guided Refinement," in *Proc. Of International Conference on Theory and Applications of Satisfiability Testing*, pp. 114–128, 2012.

[25] B. Alizadeh and M. Fujita, "Modular Datapath Optimization and Verification Based on Modular-HED," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, no. 9, pp. 1422-1435, 2010.

Bijan Alizadeh (SM'13) received his B.Sc. and M.Sc. degrees in Electrical and Computer Engineering from the University of Tehran, Iran in 1996 and 1999, respectively, and completed his Ph.D. at the University of Tehran, Iran in 2004. He then worked in the School of Electrical Engineering at the Sharif University of Technology, Iran as an assistant professor from 2005 to 2007 and joined the VLSI Design and Education Center (VDEC), The University of Tokyo, Tokyo, Japan as a research associate from 2007 to 2010. Since 2011, he works as an assistant professor at the School of Electrical and Computer Engineering (ECE) at the University of Tehran, Iran, where he established the Design, Verification and Debugging of Embedded Systems Laboratory (DVDESlab).
Dr. Alizadeh has been engaged in the research and development of VLSI systems, reconfigurable computing, formal verification, post-silicon debug and high level synthesis (HLS). He has over 85 publications in international scientific journals and conferences. He is a senior member of IEEE.

Seyyed Reza Sharafinejad received his BSc and MSc degrees in electrical and electronic engineering from Shahid Beheshti University, Tehran, Iran, in 2009 and 2011, respectively. He is currently working toward the PhD degree at the University of Tehran, Tehran, Iran. His primary research interests are low power verification, debugging of digital systems and embedded system design methodologies.

AUTHOR'S RESPONSE

Dear Editor and Reviewers,

We would like to thank the Associate Editor for the time and effort in administrating the review of this manuscript, and the anonymous reviewers for their valuable comments and suggestions to further enhance the presentation of the manuscript. We have tried to address the problems in the manuscript pointed by the reviewers separately and revised accordingly. Based on all the comments, we have appropriately revised the manuscript: i) the correctness of the proposed method is formally proved; ii) more experiments are added to the revised manuscript; iii) the typographical errors are corrected. Please note that all major changes in the revised manuscript are highlighted in red color.

Response to Reviewer 1's comments

Q1: The authors present an approach for correction of gate-level design. The basic approach works computing two solutions (i.e. corrections) of the circuit using a gate-replacement fault model. Given this two solutions they compute a new test case which shows the different behavior of both solutions. A developer has to decide which is the correct result for this new test case (i.e. he writes the corresponding test oracle) and consequently at least one of the resulting solutions is discarded. This is repeated until no difference between solutions can be found, or no new solution is left.
The approach is described very well and can be easily been understood. The English is good.

[Response]
We thank the reviewer for the positive comments and the concise summary of the novelty, merits and significance of our contribution.

Q2: Minor remark: On page 3 the authors claim that "this process[...] generate as small as possible test patterns" which is obviously wrong and in the later the authors also only claim small test sets are created.

[Response]
We have made a slight modification to the related sentences in the revised manuscript ("more compact test patterns" instead of "as small as possible test patterns").

Q3: Unfortunately, I believe that the presented algorithm is incorrect. The problem is with different but equivalent solutions. The authors stop if they cannot find any different behavior between two solutions and claim that both are correct. This assumption however is incorrect. For this please see the attached file. Let (1) be the original design and (4) be the correct one. The test case is (11/0). Assume further the returned results would are (2), (3), obviously they are equivalent and therefore the approach would consider them as correct. However, they are no correction. Therefore in case two solutions are found to be identical, both have to be block and the approach must search for further solutions. This example may look small but more complex examples can also be created in the end it just too equivalent but not equal circuits. As this change most likely effects the runtime the evaluation has to be repeated.

[Response]
We greatly appreciate the effort of the reviewer in drawing gate level examples to clearly illustrate special cases which are not supported by our proposed correction method. In order to cover this case, we have modified the proposed algorithm and revised the text accordingly. Please see ALGORITHM3 and related

texts on pages 5 and 6 of the revised manuscript. Please note that Section III is completely revised so that ALGORITHM1 in the previous version of the manuscript is now ALGORITHM3. Also, correctness proof is added to the revised manuscript (see subsection III-E). Moreover, the results have been updated (please see Tables 2-4 and Figures 5-9 on pages 7 to 10).

Q4: Additionally, the authors should point out that if the circuit is not correctable with the given the existing gates, it may return incorrect results. Let us assume the equivalence problem is fixed as described above. Again consider the example but this time the result should constant zero. The solutions will be (2/3) and (4) the resulting test case may be 10/0 or 01/0, which kills solutions (2/3). Then no other nonequivalent solutions exist for the two test cases and (4) is returned, which is incorrect.

[Response]
It is correct that if the circuit is not correctable with the existing gates, it may return incorrect results. We have added some sentences to the revised manuscript to point out this issue. Please see Section III of the revised manuscript.

Response to Reviewer 2's comments

Q1: This work addresses a well-known and extremely relevant problem with an interesting solution. The presentation of the contributions is reasonable at a high level, but the manuscript itself is peppered with areas that should be improved.

[Response]
We thank the reviewer for the positive comments.

Q2: Using a miter (or any XOR gate, for that matter) has a tendency to severely degrade runtime performance for solving the CNF formula. Your interpretation of the experimental results seems optimistic on your method's runtime. It's not obvious how or why the number of potential bug locations translates to the runtime performance of the test generation specifically. The biggest indicator that your methodology performs well would be a combination of large design, long error trace, and multiple test patterns generated. The experiments should be expanded to include some form of profiling in this direction (suggestion: profile number of test patterns generated vs runtime of the test pattern generation).

[Response]
By increasing the number of potential bug locations, the number of correction multiplexers should be added to the enriched model increases as well. It causes the CNF formula size to grow which affects the runtime of the test pattern generation process. To clarify it, some sentences are added to the revised manuscript in subsection IV-B.
On the other hand, what the reviewer suggested is also interesting and therefore we have added a new experiment to the revised manuscript (see Experiment 5 and Fig. 8 on page 9) to profile the number of test patterns generated versus the runtime of the process of test pattern generation.

Q3: On page 4, a set of primitive logic gates is listed. Are there any (primitive) logic gates for which your methodology does *not* work? If not, make this clear. If so, explain why.

[Response]

We assume that the RTL design is synthesized based on a library containing two-input primitive gate types AND, NAND, OR, NOR, XOR and XNOR as well as NOT and BUFF. It is obvious that if, for example, we have three-input gates in the gate level circuit, before applying our method, such gates need to be converted to two-input ones. Otherwise, our method would not be applicable. Some sentences are added to the revised manuscript to point out this issue. Please see the last paragraph on page 3 (subsection III-A).

Q4: "Bug-free" is an expression that should not be used. It is too strong a claim. Use a more diplomatic expression, such as "does not exhibit erroneous behavior", or something to that effect. Bug free is relevant when one can deduce 100% coverage.

[Response]
The proposed expression is used.

Q5: Use math mode for variables and inline equations or expressions consistently. For example, when test vectors are first introduced, the variables are formatted incorrectly as text. Later, select signals are written in plain text with italics instead of using math mode. Later still, test vectors are again text, but this time with italics. Just use LaTex math mode for all.

[Response]
We have used math mode for variables and inline equations or expressions.

Q6: Use LaTex math mode for (numbered) equations. The equations in this manuscript use a slightly different typeface upon close inspection. At the very least, use a serif typeface for algorithms. Ideally, use some algorithm package in LaTex. Algorithm 1 inconsistently switches between a serif and sans serif typeface.

[Response]
They have been revised accordingly.

Q7: Figure 1(b): The lines of the bubble intersect with the wires, making it a little hard to see what is going on. Figure 7: Same as Figure 1(b).

[Response]
They have been redrawn.

Q8: After Figure 3, lines in Algorithm start being referenced. Given the distance between this paragraph and the initial reference of Algorithm 1, you should make clear that these lines refer to Algorithm 1 and not Algorithm 2 to the immediate right.

[Response]
It is done.

Q9: Experiments: use the present tense throughout this section, rather than past tense. The lines on the plots are too wide. They should be a just a little thinner for improves readability. The caption of Table 2 is too long, for several reasons. 1) Including units is redundant, as they are labelled in the appropriate columns. 2) Describe "N.A.", "TO", and the timeout limit when explaining the table, rather than in the caption.

[Response]
These suggestions are taken into account in the revised manuscript.

Q10: Minor writing corrections:

[Response]
We thank the reviewer for the time and effort. Typos are corrected.

Response to Reviewer 3's comments

Q1: I like the basic idea of the method, experiments are convincing to a large extent. However, I have major concerns with the manner in which the method is presented.

[Response]
We thank the reviewer for the positive comments.

Q2: Examples are provided to demonstrate the flow of the method. Examples are helpful, but they are not sufficient, I miss a solid proof of the correctness of the approach.

[Response]
The correctness of the proposed method has formally been proved in the revised manuscript (Theorems 1 and 2 on pages 5 and 6). Please see Section III (pages 3-6) which is completely revised to address this issue as well.

Q3: What are the assumptions when starting the procedure? (E.g., is it sufficient to have just one input pattern that has to be corrected at the beginning?)

[Response]
Although we have taken enough test patterns into account in the experiments, we actually have no assumption on the number of test patterns when starting the proposed method. Having enough test patterns when starting the method makes it quickly converge to the final solution. Having just one input test pattern at the beginning, however, needs much more run time to converge the final solution due to spurious solutions which should be removed. More sentences are added to the first paragraph of Section III in the revised manuscript on page 3 to clarify it.

Q4: Why does the corrected circuit behaves correctly at the end of the procedure? I miss a correctness proof. Since this is the crucial point of the whole approach, this is absolutely necessary.

[Response]
The correctness of the proposed method has been formally proved in the revised manuscript (Theorems 1 and 2 on pages 5 and 6). Please see Section III (pages 3-6) which is completely revised to address this issue.

Q5: It is not clear that the method really works in the sequential case. Something has to be stated with respect to the number of unrollings. If the authors want to keep the title more details on the sequential case (varying unrolling depth, etc ...) have to be given. Otherwise, the title is kind of misleading.

[Response]
To cover this case, Experiment 6 and Fig. 9 are added to the revised manuscript on page 9.

Q6: equation (4) is hard to understand in the way it is written. The explanation below equ. (4) has to be more detailed.

[Response]
Section III has completely been revised. More explanations are added to the revised manuscript on page 4.

Q7: the miter construction (e.g. Fig 1 and Fig, 6) is described numerous times in the paper, this is not necessary and should be substituted by a more detailed description (including proofs) of what is novel in the approach.

[Response]
Fig. 6 and Fig. 7(b) are removed. Section III has completely been revised so that more explanations including proofs are added to the revised manuscript on pages 3-6.

Q8: what is the reason for selecting just the circuits given in Table 1?

[Response]
The main reason for selecting the circuits listed in Table 1 is the fact that these circuits are almost standard circuits acceptable to the verification/debug community. We have tried to select them in such a way that combinational and sequential circuits with different sizes (from small to very large) are available so that we can discuss the scalability of our correction method.

Best regards,
Bijan Alizadeh and Seyyed Reza Sharafinejad