Automated Design Debugging in a Testbench-Based Verification Environment

Mehdi Dehbashi André Sülflow Görschwin Fey
Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{dehbashi, suelflow, fey}@informatik.uni-bremen.de

***Abstract*—Debugging is one of the major bottlenecks in the current VLSI design process as design size and complexity increase. Efficient automation of debugging procedures helps to reduce debugging time and to increase diagnosis accuracy. This work proposes an approach for automating the design debugging procedures by integrating SAT-based debugging with testbench-based verification. The diagnosis accuracy increases by iterating debugging and counterexample generation, i.e., the total number of fault candidates decreases. The experimental results show that our approach is as accurate as exact formal debugging in 71% of the experiments.**

***Keywords*—automated debugging; testbench; diagnostic trace;**

I. Introduction

The size and complexity of VLSI designs has increased significantly during the recent years. In this situation, the debugging process is a major bottleneck in the design flow. Once the verification tool detects a design error, the error is returned as a counterexample which shows a misbehavior in the design. Having a counterexample, localization and rectification of the erroneous behavior, i.e. debugging, remains often as a manual task that consumes a significant portion of design time. Thus, automated debugging approaches are necessary to speed up the process. Among these approaches, debugging based on *Boolean Satisfiability* (SAT) [1] has been shown as a robust and efficient approach in a variety of design scenarios from diagnosis to debugging properties. The purpose of SAT-based debugging is to automatically identify the potential sources of an observed error by using the available counterexamples. Each potential source of the error is returned as a fault candidate which is a set of components of the circuit. Each fault candidate can fix all erroneous behavior of counterexamples using non-deterministic replacements.

Different approaches have been proposed for enhancing the performance and accuracy of SAT-based debugging. They can be categorized into two groups. The approaches of the first group try to enhance the debugging performance with a given set of counterexamples. The approaches of the second group generate more counterexamples for improving the accuracy of the debugging process.

The approaches based on the given set of counterexamples aim at reducing run time and memory requirements. The work in [2] exploits the hierarchical nature of modern designs to improve the performance and quality of debugging. That work as well as [3] uses *Quantified Boolean Formulas* (QBF) to reduce the size of the problem instance. In [4], *Maximum SATisfiability* (MaxSAT) [5] is used to improve the performance and applicability of debugging. MaxSAT allows for a simple formulation of the debugging problem and therefore reduces the problem size and run time. Abstraction and refinement techniques are used in [6] for handling large designs with a better performance and reduced memory consumption. Totally the main drawback of these approaches is that the diagnosis accuracy is limited by the given set of counterexamples.

The approaches of the second group combine counterexample generation (verification) and debugging in a single flow. With each new counterexample, the *diagnosis accuracy* increases by excluding fault candidates that cannot fix erroneous behavior of the added counterexample. Thus, a high quality counterexample is a counterexample which reduces the number of fault candidates effectively. The work in [7] uses randomly generated counterexamples for debugging and applies automatic correction based on re-synthesis. Automatic correction increases the computational costs and is not guaranteed to fix an error in the desired way. Using random counterexamples may decrease the diagnosis accuracy, and may increase the iterations between verification and debugging. In [8], a heuristic approach based on three-valued logic is used to find high quality counterexamples. For an injected X value at a fault candidate and by observing X values on outputs, the approach assumes that modifying the fault candidate can create any value at the output. But this is an over-approximation due to the conservative properties of X values. An exact approach based on QBF is proposed in [9]. That creates high quality counterexamples to find fault candidates fixing any erroneous behavior. However, the explicit enumeration of fault candidates may decrease the debugging speed for large designs. The approaches in [8] and [9] need a formal specification for creating high quality counterexamples. However, a formal specification is often not given for complex designs. Here a testbench is used to create the expected output response of an input stimulus.

In this paper, we present a flow to improve the accuracy of SAT-based debugging when a testbench is used for verification. No formal specification is required. At first *diagnostic traces* are derived from the faulty implementation. A diagnostic trace is an input stimulus which tries to activate a fault candidate (or a set of fault candidates) and to propagate its behavior to the outputs. The diagnostic traces help testbench-based verification to create high quality counterexamples. Then these counterexamples are used for iterating SAT-based debugging and increasing the diagnosis accuracy. The techniques in this paper do not need a fault model for generating the diagnostic traces. Moreover, diagnostic traces are created by way of the faulty implementation and the initial set of fault candidates only. Whereas the focus of this paper is on designs using a testbench, these diagnostic trace generation techniques

This work has been funded in part by the German Research Foundation (DFG, grant no. FE 797/6-1).

can also be applied for debugging designs using a formal specification. Experimental results compare three heuristics to find diagnostic traces by run time, memory, and accuracy to random trace generation and to the formal approach of [9] that is exact but requires formal specification. The experimental results show that our approach is as accurate as exact formal debugging in 71% of the experiments.

The remainder of this paper is organized as follows. Section II introduces preliminary information on three-valued logic, circuits and sensitized paths, and SAT-based debugging. Then, our approach is presented in Section III. Section IV presents experimental results on benchmark circuits. The last section concludes the work.

II. Preliminaries

A. Three-Valued Logic in Boolean Satisfiability

In three-valued logic, each signal can have the value 0, 1, or X (unknown). This logic has been used in the field of formal hardware verification for creating strong counterexamples [10], [11] and faster verification engines [12], [13]. Also, this logic has been used for generating high quality counterexamples [8].

An encoding in *Conjunctive Normal Form* (CNF) is needed for the three-valued logic defined over {0, 1, X} to apply a standard SAT solver. Accordingly, the modeling of gates and components in the CNF formula has to be adjusted. Here, three-valued logic is encoded by using two variables for each signal similar to [12]. The three-valued constants 0, 1, and X are defined by pairs $(0, 0)$, $(0, 1)$ and $(1, -)$, respectively.

B. Circuits and Sensitized Paths

Each combinational circuit is represented by a directed acyclic graph $C = (V, E)$, referred to as the *circuit graph*, where V is the set of circuit nodes and E, the set of edges, corresponds to the gate input-output connections in the circuit [14]. The *fan-out* of v is a set of nodes u, such that there is an edge from v to u. The *fan-in* of v is a set of nodes w, such that there is an edge from w to v. A path P from node g_0 to node g_r is a sequence of nodes, $(g_0, g_1, g_2, \ldots, g_r)$ such that $(g_{i-1}, g_i) \in E$. An edge whose value changes under the presence of some fault(s) is called a *sensitized edge*, and a path of sensitized edges is called a *sensitized path* [15]. In this paper, an X value is injected at a fault candidate to over-approximate a sensitized path. An input to a node is said to have a *controlling value (cv)* if it determines the value of the node output regardless of the values on the other inputs to the node. If the value on some input is the complement of the cv, the input is said to have a *non-controlling value (ncv)*. An input with X value is neither a cv nor a ncv.

C. SAT-based Debugging

Debugging is a procedure in a design process that is started when the implementation of the design has failed verification. The output of the verification engine is typically returned as a set of counterexamples CEs which proves the existence of a bug in the implementation. A counterexample $CE_i \in CEs$ usually includes input stimuli causing erroneous behavior, and the expected correct output response.

An approach for SAT-based debugging was presented in [1] that searches for all possible fault candidates in the implementation. Given an implementation of a circuit and a set of counterexamples, one copy of the circuit is created for each counterexample. Then, the inputs and outputs are constrained to the input stimuli and to the correct output response of the corresponding counterexample. Also the circuit is enhanced with correction logic by adding a multiplexer at the output of each component. The original output function F_c of component C is replaced by F'_c. The select line S_c of the added multiplexer controls F'_c such that if S_c is activated $F'_c = R_c$ where R_c is an unconstrained variable and a value for correcting the erroneous behavior may be injected, otherwise $F'_c = F_c$. The select line is also called *abnormal predicate*. The number k of active abnormal predicates is controlled by a fault cardinality constraint.

Debugging for sequential circuits is done by unrolling the circuit for some time steps equal to the length of the counterexample [16]. The correction logic is added as in the combinational case and usually the same abnormal predicate is used for the same gate in all time steps and for all counterexamples.

Conceptually, for each counterexample CE_i there is a set of fault candidates $\mathcal{F}_{CE_i}$. For single faults these sets are intersected to return the final set of fault candidates $\mathcal{F}$:

$$\mathcal{F} = \mathcal{F}_{CE_1} \cap \mathcal{F}_{CE_2} \cap \ldots \cap \mathcal{F}_{CE_n} = \bigcap_{i=1}^{n} \mathcal{F}_{CE_i} \tag{1}$$

For counterexample generation, the goal is to minimize the size of $\mathcal{F}$ and the number of counterexamples n in Formula (1). The set $\mathcal{F}$ shows the diagnosis accuracy, and parameter n is effective on the memory and the time of debugging engines. Thus, an algorithm which can minimize the number of fault candidates with a minimum number of counterexamples in a reasonable time improves the performance of the debugging engines.

For multiple faults, each fault candidate $FC_i \in \mathcal{F}$ includes a tuple of k components, $FC_i = \{FC_{i,1}, FC_{i,2}, \ldots, FC_{i,k}\}$. The set $\mathcal{F}$ is constituted in a way that each component of fault candidate FC_i is contained by at least one sensitized path of one counterexample.

Fault model-free SAT-based debugging does not need a fault model. As a drawback fault masking may not be recognized. This case occurs for multiple faults when one fault masks behavior of another fault. This is a known problem but not addressed in this work.

III. Integration of Formal Debugging with Testbench-Based Verification

This section presents the approach to combine SAT-based debugging and counterexample generation which is aided by generating diagnostic traces. In this approach, the diagnostic traces help to create high quality counterexamples for automated design debugging to increase the diagnosis accuracy.

Figure 1 shows the overall approach which consists of three main steps. These steps are debugging, diagnostic trace generation, and running the testbench to validate diagnostic traces. Debugging is started by having the design and an initial counterexample for finding all fault candidates which can correct the erroneous behavior of the circuit exhibited by the initial counterexample.

The second step of the approach, called diagnostic trace generation, is the main focus of this work. The inputs of this

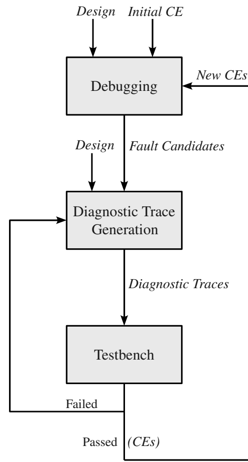

Fig. 1. Integration of debugging and testbench-based verification

step include the faulty design and the set of fault candidates. The aim of this step is to generate diagnostic traces by heuristic methods. As a high quality counterexample aims at reducing the number of fault candidates effectively, each diagnostic trace activates a small number of fault candidates and observes their behavior on outputs. Therefore, the counterexample derived from diagnostic traces can likely reduce the number of fault candidates.

Afterwards, the diagnostic traces are tested in a testbench environment, because it is not guaranteed that the diagnostic traces really create erroneous output responses in the design. Here, a testbench is used as a black box specification for creating the expected correct output of a diagnostic trace. A diagnostic trace creating erroneous output behavior is a counterexample and the step validates the diagnostic traces. If there is no counterexample, the algorithm returns to the second step for generating more diagnostic traces. Otherwise, the algorithm continues debugging with the new counterexamples.

In the following, Section III-A describes the intuition behind the diagnostic traces to create high quality counterexamples. Sections III-B, III-C, and III-D introduce three heuristics for diagnostic trace generation. The discussion of each section starts with a single fault assumption, then it is followed by an extension to multiple faults.

A. Counterexample versus Fault Candidate

This section describes why more counterexamples are effective for reducing the number of fault candidates and how we can derive a high quality counterexample. Figure 2(a) shows a faulty circuit with a single fault. The real fault location is shown by a circle. The counterexample CE_1 is propagated through the dashed path to the circuit output $O1$. The fault candidates indicated by $\times$ can correct the erroneous behavior of $O1$. Actually, the fault candidates show the sensitized path related to the counterexample. The set of fault candidates related to CE_1 is written as $\mathcal{F}_{CE_1}$. Figure 2(b) shows the effect of the second counterexample CE_2 separately. The counterexample CE_2 is propagated through another path to $O2$ and creates $\mathcal{F}_{CE_2}$. The effect of using both CE_1 and CE_2 in the debugging procedure is described in Figure 2(c). According to Formula (1) , the number of fault candidates is reduced to the set of $\mathcal{F} = \mathcal{F}_{CE_1} \cap \mathcal{F}_{CE_2}$ where each fault candidate $FC_i \in \mathcal{F}$ is a single fault candidate. Now it is

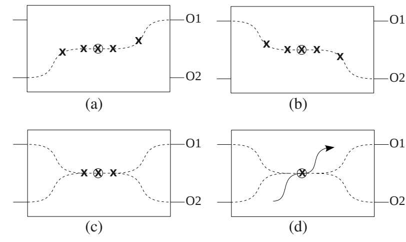

Fig. 2. Counterexample versus fault candidate: (a) Fault candidates of CE_1. (b) Fault candidates of CE_2. (c) Fault candidates of CE_1 and CE_2. (d) Fault candidates of CE_1 and CE_2 and CE_3

interesting to figure out how a high quality counterexample can have the strongest effect in reducing the number of fault candidates. Figure 2(d) shows the counterexample CE_3 which further reduces the size of $\mathcal{F}$. The sensitized path of CE_3 has the minimum intersection with the sensitized paths of other counterexamples.

For multiple faults, each fault candidate FC_i can have k components. In this case, the sensitized paths of a new counterexample intersect with a fault candidate FC_i when all of the sensitized paths leading to erroneous behavior propagate through the components of FC_i. Thus, a new counterexample which has minimum intersection with fault candidates can effectively reduce the size of $\mathcal{F}$.

B. Local Branch Activation (LBA)

In this technique, the X value is considered as a token for the behavior of one fault candidate and the algorithm tries to propagate the token X through different branches around a fault candidate to the primary outputs, i.e., different sensitized paths are activated for each fault candidate. By activating different sensitized paths around a fault candidate, the intersection with paths related to other fault candidates is likely reduced, because the fault candidates are usually adjacent and close to each other in the circuit graph. As a result, the created counterexample usually decreases the number of fault candidates in the debugging procedure. In the following, first the branch and path activation method is described and then the total algorithm is presented.

1) Branch and Path Activation: Considering the node g_{FC}, traversing the circuit graph from node g_{FC} along successor nodes until reaching a fan-out, is called *forward branch* for g_{FC}. Traversing the circuit graph from node g_{FC} along predecessor nodes until reaching a fan-in, is called *backward branch* for g_{FC}. Before reaching the forward branches for g_{FC}, there are some middle nodes which are collected in the set MG, and the nodes of forward branches are collected in the set FG. Figure 3 shows an example of a circuit graph. Considering g_{FC}, the sets $MG = \{g_{mf_1}, g_{mf_2}\}$ and $FG = \{g_{f_1}, g_{f_2}\}$ result. Thus the number of forward branches is $|FG| = 2$. The nodes of backward branches are collected in the set BG. The set BG for g_{FC} in Figure 3 is $BG = \{g_{b_1}, g_{b_2}\}$ and thus the number of backward branches is $|BG| = 2$. The X value of g_{FC} can be propagated through $|BG| \times |FG|$ different paths. The method activates the individual paths to more likely reduce the intersection with paths related to other fault candidates. As shown in Figure 3, one of the paths is $P = (g_{b_2}, g_{mb_2}, g_{mb_1}, g_{FC}, g_{mf_1}, g_{mf_2}, g_{f_1})$. For propagating the X value of g_{FC} through the path P, the following constraints are required:

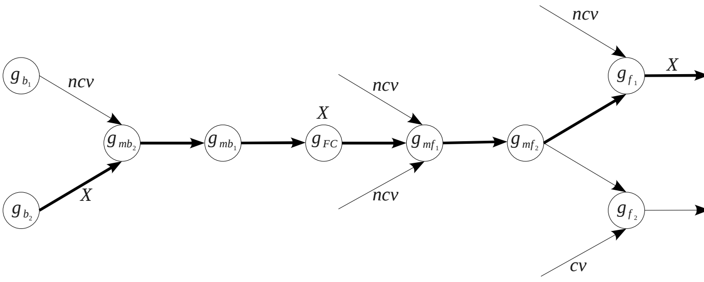

Fig. 3. Branch and path activation

- The off-path inputs of each gate $g_{mf_i} \in MG$ have to have a ncv.
- The off-path inputs of each gate $g_{f_i} \in FG$, $g_{f_i} \notin P$ have to have a cv.
- The off-path inputs of each gate $g_{f_i} \in FG$, $g_{f_i} \in P$ have to have a ncv.
- The output of each gate $g_{b_i} \in BG$, $g_{b_i} \notin P$ has to have a ncv.
- The output of each gate $g_{b_i} \in BG$, $g_{b_i} \in P$ has to have an X value.

2) LBA algorithm: The algorithm of the complete procedure is described in Figure 4. The algorithm data are a faulty design D and an initial counterexample CE_1 (line 1). The initial counterexample is assigned to the set of counterexamples CEs (line 2). The first step is SAT-based debugging (line 7). SAT-based debugging is responsible to find all fault candidates which can rectify the erroneous behavior of counterexamples (CEs). Then for each fault candidate FC, the function LBA generates some diagnostic traces (line 10), after that the diagnostic traces are checked by the testbench to detect whether they produce a counterexample (line 11). If at least one new counterexample is found, the algorithm continues the debugging step for the new set of counterexamples. The algorithm finishes when there is no new counterexample for any existing fault candidates.

Figure 5 shows the function LBA which is responsible for generating diagnostic traces for a fault candidate. After converting the faulty design to CNF, additional constraints are inserted in the CNF. The constraint of line 2 assigns the value X to a fault candidate, and line 3 causes that X to be observed at least at one primary output (PO). Then the algorithm searches for the backward and forward branches considering one fault candidate, and constitutes different paths for X propagation (line 4). For each path, the appropriate constraints as mentioned in the Section III-B1 are applied (line 8). Then the SAT solver searches for a solution that is added to the set of diagnostic traces (lines 9-10).

In Figure 5, there are three types of constraints: fault candidate constraint, observability constraint (primary outputs constraint), and path constraint. For inserting a new constraint, firstly any previous constraint of the same type is removed, then the new constraint is inserted. For simplicity it is not shown in Figure 5.

Each of the additional constraints consumes some memory which can be measured by the number of clauses. One unit-literal clause is added per fault candidate constraint. For each path, if standard gates are used in the circuit, two unit-literal clauses for backward branches, m unit-literal clauses

```
1  function Debugging_LBA(D, CE_1)
2  CEs = CE_1
3  New_CEs = ∅
4  do
5  {
6     CEs = CEs ∪ New_CEs
7     F = SAT_Based_Debugging(D, CEs)
8     foreach Fault Candidate FC ∈ F do
9     {
10       Diag_Traces = LBA(D, FC)
11       New_CEs = Testbench(Diag_Traces)
12       if New_CEs! = ∅ then break
13    }
14 } while New_CEs! = ∅
15 end function
```

Fig. 4. Automated debugging using LBA method

```
1  function LBA(D, FC)
2  Constraint(FC = X)
3  Constraint(∑_{i=1}^{q}(PO_i == X) ≥ 1)
4  Paths = Find_Branches(FC)
5  Diag_Traces = ∅
6  foreach Path ∈ Paths do
7  {
8     Constraint(Path)
9     if Solve() == SAT then
10       Diag_Traces = Diag_Traces ∪ Extract_Trace()
11 }
12 end function
```

Fig. 5. LBA algorithm

for middle nodes ($|MG| = m$), and two unit-literal clauses for forward branches are added. For the observability constraint, $O(q)$ clauses are inserted [1], where q is the number of primary outputs. Thus, the number of additional clauses are $1 + (2 + m + 2) + O(q) = O(m) + O(q)$.

Fault ranking techniques [14] may help increasing the performance of this algorithm. If a fault candidate related to the real bug location is processed earlier, then the diagnostic traces create at least one counterexample with higher probability.

For sequential circuits, firstly the circuit is unrolled for some time steps. In this case, each time step is like a combinational circuit and branches and paths are activated like in the combinational case. A fault candidate has one component in each time step. For applying the LBA method to sequential circuits, we activate one fault candidate and its paths in each time step independently. Then the algorithm investigates whether the X can be observed on outputs. For multiple faults, where a fault candidate includes multiple components, we activate each component independently. Therefore the memory consumption for additional constraints remains as mentioned before.

C. Minimization of Sensitized Path Intersection (MSPI)

This technique finds the sensitized paths including a minimum number of existing fault candidates. Again, the token X is used for the behavior of one fault candidate or a set of fault candidates. The token X is propagated from inputs, crosses a number of fault candidates and arrives at outputs. The number of fault candidates having value X is denoted by L:

$$\sum_{i=1}^{|\mathcal{F}|}(FC_i == X) = L \tag{2}$$

```
1   function Debugging_MSPI(D, CE_1)
2   CEs = CE_1
3   New_CEs = ∅
4   L = 1
5   do
6   {
7      CEs = CEs ∪ New_CEs
8      F = SAT_Based_Debugging(D, CEs)
9      New_CEs = ∅
10     while L < |F| and New_CEs == ∅ do
11     {
12        Diag_Traces = MSPI(D, F, L)
13        New_CEs = Testbench(Diag_Traces)
14        L = L + 1
15     }
16  } while New_CEs! = ∅
17  end function
```

Fig. 6. Automated debugging using MSPI method

```
1   function MSPI(D, F, L)
2   Constraint(∑(FC_i == X) = L)
3   Constraint(∑(PO_i == X) ≥ 1)
4   Diag_Traces = ∅
5   VisitedFCs = ∅
6   while |VisitedFCs| < |F| do
7   {
8      NonVisitedFCs = F \ VisitedFCs
9      Constraint(∑(NonVisitedFC_i == X) ≥ 1)
10     if Solve() == unSAT then break
11     Diag_Traces = Diag_Traces ∪ Extract_Trace()
12     VisitedFCs = VisitedFCs ∪ Extract_VisitedFCs()
13  }
14  end function
```

Fig. 7. MSPI algorithm

The technique starts with $L = 1$ to find paths sensitizing one fault candidate. Thus the token X is propagated from inputs, crosses one fault candidate, and arrives at outputs. If there is no path with one fault candidate having X which can create a diagnostic trace or a counterexample, the paths with more fault candidates having X are searched until at least one counterexample is found. The algorithm continues until L is equal to the number of fault candidates.

Figure 6 shows the algorithm. Line 2 assigns the initial counterexample to the set of counterexamples CEs. The set of new counterexamples New_CEs and L are initialized in lines 3-4. The first step is done by SAT-based debugging and extracts the fault candidates $\mathcal{F}$ (line 8). While L is less than the number of fault candidates and the set New_CEs is empty (line 10), the MSPI function searches for diagnostic traces (line 12). In the next step the testbench checks the diagnostic traces for generating new counterexamples (line 13). In line 14, L is increased. If the diagnostic traces yield at least one new counterexample, the algorithm continues with the debugging step. Otherwise the algorithm searches for new paths with more fault candidates having X. The algorithm finishes when L and the number of fault candidates converge.

The MSPI function is described in Figure 7. Firstly the faulty design is converted to CNF. Then additional constraints are inserted into the CNF. Line 2 determines the number of fault candidates having an X value. This number is specified by L. Line 3 applies the observability constraint in order to see the erroneous behavior at least on one output. At this point, different methods can be applied for finding the diagnostic traces. One method can be simply finding all existing diagnostic traces according to the applied constraints (without need to lines 5-13). The weakness of this method is that usually there are many solutions with respect to L. Thus, the algorithm performance is biased by increasing the run time significantly. To overcome this weakness, the number of extracted diagnostic traces for each L should be limited by applying some heuristics. Lines 5-13 apply a heuristic method. When a new diagnostic trace is found, the fault candidates having an X value on this diagnostic trace are added to the set of $VisitedFCs$. Next time, the algorithm searches for a diagnostic trace that includes at least one non-visited fault candidate (lines 8-9). By this, more fault candidates are covered by the diagnostic traces.

When the number of diagnostic traces in MSPI is limited, some counterexamples may not be found by the testbench. To overcome this weakness, another method is applied. When the MSPI algorithm is finished, there typically remains a small set of fault candidates. In this case, executing MSPI for another round may find the missed counterexamples.

In Figure 7, for applying the constraint related to the heuristic method (line 9) and the constraint on fault candidates (line 2), $O(f)$ clauses are added to the CNF, where f is the number of fault candidates. Also $O(q)$ clauses are needed for the obervability constraint on primary outputs (line 3). In total, $O(f) + O(q)$ additional clauses are needed for combinational circuits with a single fault.

For multiple faults, each fault candidate $FC_i \in \mathcal{F}$ can have k components: $FC_i = \{FC_{i,1}, FC_{i,2}, \ldots, FC_{i,k}\}$. Also the erroneous behavior of a fault candidate can be propagated to outputs by each component or by a combination of components. This behavior is modeled by the following formula:

$$FC_i = \bigvee_{j=1}^{k} (FC_{i,j} == X) \tag{3}$$

Thus for each FC_i, the X-behavior can be observed by assigning at least one of its components to X. For sequential circuits with a single fault, a similar strategy is applied. In this case each $FC_i \in \mathcal{F}$ has one component in each time step: $FC_i = \{FC_i^1, FC_i^2, \ldots, FC_i^s\}$, where s is the number of time steps. The activation of a fault candidate in each time or its activation by a combination of times may lead to the erroneous behavior on outputs. The following formula describes this behavior:

$$FC_i = \bigvee_{t=1}^{s} (FC_i^t == X) \tag{4}$$

Now, when there are sequential circuits with multiple faults, each fault candidate has two dimensions, one dimension represents the location and one dimension represents the time. In this case firstly in each time step, the sensitized components of a fault candidate are abstracted by Formula (5):

$$FC_i^t = \bigvee_{j=1}^{k} (FC_{i,j}^t == X) \tag{5}$$

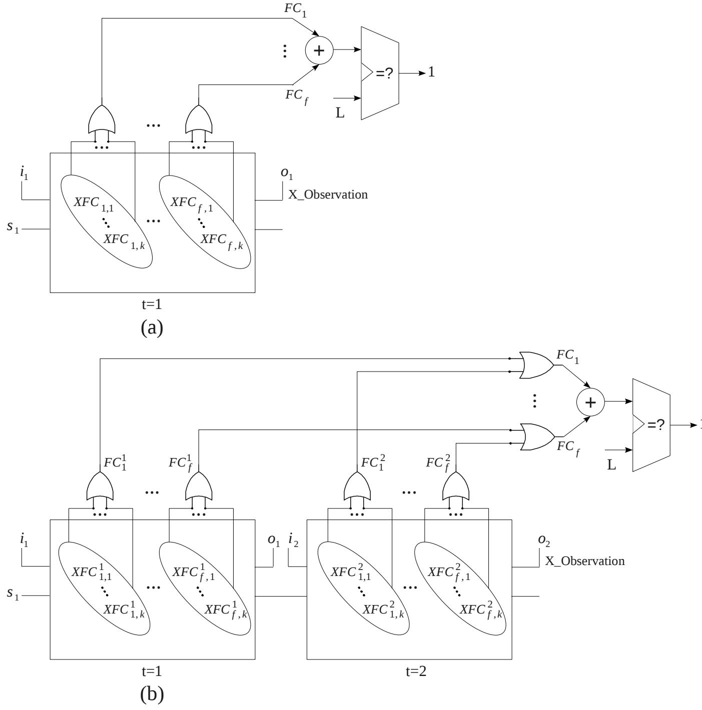

Fig. 8. MSPI method for sequential circuits with multiple faults: (a) first time step. (b) second time step

After that, the behavior of the fault candidate in time is abstracted according to Formula (6):

$$FC_i = FC_i^1 \vee FC_i^2 \vee \ldots \vee FC_i^s = \bigvee_{t=1}^{s} FC_i^t \tag{6}$$

Finally, for all of the above mentioned cases Formula (7) is used to apply the limitation (L) to all fault candidates.

$$\sum_{i=1}^{|\mathcal{F}|} FC_i = L \tag{7}$$

By using this method, the algorithm in Figure 7 can be applied for all cases.

For clarifying the mentioned formulas, Figure 8 shows a sequential circuit with multiple faults. Figure 8(a) considers one time step (this case is similar to combinational circuits with multiple faults). One OR-gate is inserted per fault candidate. The inputs of the OR-gates correspond to the variables of fault candidates specifying the X values. The outputs of the OR-gates are added and constrained to L. Also the observability constraint is applied to the primary outputs. Figure 8(b) shows two time steps. In addition to the OR-gates inserted for multiple faults, one OR-gate is applied for each fault candidate to control the fault candidate's behavior in the time dimension.

When considering $i \in [1, f]$ as fault candidate index, $j \in [1, k]$ as location index, and $t \in [1, s]$ as time index, the MSPI technique for sequential circuits with multiple faults needs $f \cdot s$ OR-gates with k inputs for each OR-gate ($f \cdot s \cdot OR(k)$) to control the locations of the fault candidates for all times. Also $f \cdot OR(s)$ are needed to control the time dimension. Thus totally this method requires $O(f) + O(q) + f \cdot s \cdot OR(k) + f \cdot OR(s)$ additional constraints where each $OR(i)$ gate has $i+1$ clauses. Therefore the number of additional clauses is in $O(f) + O(q) + f \cdot s \cdot (k+1) + f \cdot (s+1) = O(f) + O(q) + O(f \cdot s \cdot k) + O(f \cdot s)$.

```
1  function Debugging_LMBA(D, CE1)
2  CEs = CE1
3  New_CEs = ∅
4  L = 1
5  do
6  {
7     CEs = CEs ∪ New_CEs
8     F = SAT_Based_Debugging(D, CEs)
9     New_CEs = ∅
10    while L < |F| and New_CEs == ∅ do
11    {
12       Diag_Traces = MSPI(D, F, L)
13       New_CEs = Testbench(Diag_Traces)
14       L = L + 1
15    }
16
17    if L == |F| then
18    {
19     foreach Fault Candidate FC ∈ F do
20     {
21        Diag_Traces = LBA(D, FC)
22        New_CEs = New_CEs ∪ Testbench(Diag_Traces)
23     }
24     L = L + 1
25    }
26
27 } while New_CEs! = ∅
28 end function
```

Fig. 9. Automated debugging using LMBA method

D. Limited Minimization followed by Branch Activation (LMBA)

The counterexamples generated by LBA and MSPI may have different characteristics. LBA investigates each fault candidate in detail whereas MSPI has an overall view of all fault candidates. The advantages of both methods are combined in one unified algorithm (LMBA) to obtain higher accuracy. The LMBA technique tries to increase the diagnosis accuracy with a reasonable overhead by improving the MSPI technique with local branch activation. After finishing the MSPI algorithm with a small set of fault candidates, activating the branches of all fault candidates can be done in a short time and without high computational cost. By spending this short time, a higher accuracy can likely be achieved.

Figure 9 shows the details of LMBA. Firstly the algorithm searches for the paths including a minimum number of Xs on the fault candidates (lines 10-15). When the convergence of L and the number of fault candidates is reached, the first step of LMBA is finished. Thus, the second step of LMBA starts. Now, the local branches of all fault candidates are activated (line 19-23) and the new counterexamples are collected (line 22). If there is at least one new counterexample then SAT-based debugging is executed. After that the LMBA algorithm finishes. The number of required additional clauses for the LMBA method is the maximum of additional clauses of LBA and MSPI:
$\max(O(m) + O(q)\ ,\ O(f) + O(q) + O(f \cdot s \cdot k) + O(f \cdot s))$.

IV. Experimental Results

In this section, the effects of the presented techniques are experimentally demonstrated. The techniques described

TABLE I
RESULTS FOR SINGLE FAULTS

Method		Heuristic Methods												Random Method				Formal Method			
Name		LBA				MSPI				LMBA				RND				QBF [9]			
Circuit	#Gates	#FC	#CE	Time	Mem	#FC	#CE	Time	Mem	#FC	#CE	Time	Mem	#FC	#CE	Time	Mem	#FC	#CE	Time	Mem
comb.																					
apex5	3938	**2**	7	19.8	39	**2**	7	12.7	35	**2**	7	13.4	35	3	21	66.4	104	2	4	22.8	26
c7552	4674	22	18	639.9	188	22	17	712.8	204	22	18	538.7	139	22	21	691.7	206	22	6	346.5	77
cordic	2938	10	18	109.5	78	10	8	46.1	39	10	12	43.7	51	10	21	260.7	102	10	3	30.6	19
dalu	2883	4	15	82.1	102	4	9	14.1	39	4	13	29.7	52	4	21	49.2	104	4	2	14.9	19
des	3942	2	3	3.7	18	2	3	6.6	26	2	3	6.5	26	2	21	70.8	138	2	2	11.1	18
i10	3294	**6**	18	886.5	104	12	11	253.8	102	7	17	360.7	103	14	21	593.2	137	6	3	52.5	20
misex3	6249	**2**	15	126.9	105	**2**	7	29.1	52	**2**	7	24.5	52	4	21	443.9	140	2	3	18.8	38
pair	2848	**9**	11	182.5	69	**9**	10	87.2	68	**9**	14	84.7	69	10	21	192.3	96	9	3	41.3	26
seq	4776	**7**	14	141.4	103	**7**	5	39.1	38	**7**	9	46	52	9	21	176.6	138	7	3	19.7	34
seq.																					
b04	821	**6**	20	115.5	105	15	20	158.6	106	15	21	146.1	106	18	21	134.8	106	6	7	57.8	47
b05	1198	2	1	0.3	9	2	1	0.3	9	2	1	0.3	9	2	21	26.1	140	2	1	4.9	25
b08	223	6	1	0.6	2	6	1	0.3	2	6	1	1.1	2	6	21	6.8	26	4	2	1.1	5
b10	260	**1**	21	26.6	51	11	14	17.3	26	10	16	18.2	26	10	21	17.6	39	1	3	2.2	8
b11	867	9	21	73.6	138	9	3	7.6	23	9	8	21.2	47	9	21	71.3	138	5	3	10.8	34
b12	1297	29	6	69.1	68	**27**	21	329.2	207	**27**	20	201.4	189	29	21	328.1	207	27	3	78.1	51
gcd	1217	7	19	150.2	188	7	21	151.8	205	7	21	128.2	157	7	21	135.7	189	7	4	81.4	50
phase_de.	1834	29	3	45.1	48	29	16	388.5	209	29	20	434.2	277	29	21	471.4	278	29	2	113.5	47

TABLE II
RESULTS FOR MULTIPLE FAULTS

Method		Heuristic Methods															Random Method					Formal Method				
Name		LBA					MSPI					LMBA					RND					QBF [9]				
Circuit	#Gates	k	#FC	#CE	Time	Mem	k	#FC	#CE	Time	Mem	k	#FC	#CE	Time	Mem	k	#FC	#CE	Time	Mem	k	#FC	#CE	Time	Mem
comb.																										
apex5	3938	1	5	14	76.5	102	3	-	19	1447.4	153	3	-	19	1444.7	153	2	45	21	250.1	154	3	60	5	1107.6	124
c7552	4674	1	4	21	260.2	189	2	88	21	1846.9	303	1	2	4	41.2	47	1	2	21	143.8	189	2	58	9	877.1	185
cordic	2938	1	9	21	77.4	102	1	6	11	24.7	51	1	6	12	22.1	39	1	9	21	65.4	102	1	3	3	7.6	19
dalu	2883	**2**	**56**	16	395.7	184	**2**	**56**	19	424.1	185	**2**	**56**	18	561.5	184	2	-	21	1270.1	379	2	56	7	194.4	124
des	3942	3	4	20	256.6	139	3	4	12	85.3	78	3	4	13	86.5	78	3	4	21	174.9	140	3	4	3	205.3	74
i10	3294	1	2	19	101.3	104	1	1	6	23.4	47	1	1	7	31.8	39	1	1	21	65.8	105	2	6	6	84.5	68
misex3	6249	2	26	21	586.1	157	**2**	**16**	20	440.4	123	2	18	21	221.1	123	2	60	21	3270.2	254	2	16	5	145.3	76
pair	2848	2	15	20	169.4	136	2	15	11	94.2	69	**3**	-	12	193.8	160	2	15	21	151.7	136	3	-	6	2453.1	158
seq	4776	2	4	21	289.5	186	**2**	**2**	11	92.2	93	**2**	**2**	21	292.4	121	1	3	21	98.6	137	2	2	5	34.4	67
seq.																										
b04	821	1	5	21	116.4	106	1	2	6	20.3	51	1	2	14	36.1	70	1	2	21	67.3	106	2	62	13	1082.6	152
b05	1198	1	5	1	4.2	9	1	5	5	20.4	47	1	5	3	11.8	26	1	5	21	41.9	140	1	5	1	11.2	23
b08	223	1	69	1	6.2	4	**1**	**58**	20	61.6	26	**1**	**58**	20	62.4	26	**1**	**58**	21	66.1	38	1	58	2	4.5	4
b10	260	2	14	11	222.7	62	2	14	17	120.1	62	2	14	8	61.4	19	2	14	21	99.5	59	2	14	6	16.3	43
b11	867	**2**	**60**	11	1053.7	215	2	66	1	32.1	47	2	66	1	87.6	47	**2**	**60**	21	781.8	251	2	60	3	223.2	467
b12	1297	1	93	1	41.9	19	1	93	12	269.2	105	1	93	12	269.1	105	1	93	21	787.7	207	1	93	1	171.2	33
gcd	1217	2	35	12	248.4	153	**2**	**28**	19	563.3	189	**2**	**28**	17	342.2	139	2	32	21	332.6	206	2	28	3	40163.1	141
phase_de.	1834	1	11	13	215.5	207	1	11	19	374.1	276	1	11	18	298.1	211	1	11	21	303.6	278	1	11	2	41.9	51

in this paper are implemented using C++ in the WoLFram environment [17] and are evaluated on combinational and sequential circuits of LGsynth93 and ITC-99 benchmark suites. The faults are randomly injected by replacing gates. For example an AND gate is replaced by an OR gate. For bounded sequential debugging, the circuits are unrolled for five time steps.

The experiments are carried out on a Quad-Core AMD Phenom(tm) II X4 965 Processor (3.4 GHz, 8 GB main memory) running Linux. MiniSAT is used as underlying SAT solver [18]. Run time is measured in CPU seconds, and the memory consumption is measured in MB.

In the experiments, we compare the methods presented in this paper (LBA, MSPI, LMBA) to a method based on random trace generation (RND) and to QBF [9]. Note that LBA, MSPI and LMBA do not have access to a formal specification but they only use a testbench as a simulation model or a black box specification. Thus the accuracy of these methods is limited. In contrast, QBF [9] uses a formal specification and can therefore achieve a higher accuracy. In these experiments all methods are limited to a maximum of ten iterations between the debugging and the verification procedures. As mentioned in Section III-C, the MSPI method may be executed for two rounds in these ten iterations. The best results among heuristics and random methods are marked bold.

Table I presents the experimental results for single faults. The table shows the final number of fault candidates ($\#FC$), the total number of counterexamples used ($\#CE$), the required run time ($Time$), and the maximum memory consumption (Mem). The experiments are started with one initial counterexample. The QBF [9] method is the only exact approach. Thus the number of fault candidates ($\#FC$) determined by this method is considered as the minimal number of fault candidates in each experiment. In the tables, a dash (-) indicates more than 99 fault candidates. The experiments of Table I show that the diagnosis accuracy of the presented methods is better than of RND. By comparing $\#FC$ columns, LBA, MSPI and LMBA have a better accuracy than RND in 7

experiments, while RND is never better than LBA or LMBA. Also for single faults, LBA has the best accuracy among the methods presented in this paper in most of the experiments. LBA is as accurate as QBF [9] in 82% of the experiments (14 experiments), while MSPI and LMBA are as accurate as QBF [9] in 71% of the experiments (12 experiments). For $i10$, LBA has a good accuracy but long run time. LMBA obtains a good accuracy with a reasonable run time by merging the advantages of LBA and MSPI. Also LMBA and MSPI are faster than LBA for combinational circuits. In sequential circuits $b04$ and $b10$, LBA finds the minimum number of fault candidates in a relatively short time.

Table II shows the experimental results for multiple faults. In this case, each fault candidate has up to k components. The debugging procedure starts with $k = 1$ and iteratively increases k until a satisfying solution is found. This yields a fault candidate which is a tuple of k components. As mentioned in Section II-C, fault masking may not be recognized here due to fault model-free SAT-based debugging.

The methods which do not have access to a formal specification have some limitations for their diagnosis accuracy. One limitation occurs when one fault (among multiple faults) always remains inactive and never appears among the fault candidates found by the counterexamples so far. This can be seen in Table II for $i10$. MSPI and LMBA localize one fault location accurately but another fault remains always inactive ($k = 1$), while QBF [9] activates all faults ($k = 2$) by comparing the faulty circuit to a complete specification.

Comparing two approaches, a larger value of k indicates better accuracy. If the value of k is equal for two approaches, a smaller $\#FC$ indicates better accuracy. In this case, MSPI and LMBA have a better accuracy than RND in 7 experiments. MSPI and LMBA are less accurate than RND in only one experiment, while in that case they use less counterexamples. MSPI and LMBA are as accurate as QBF [9] in 59% of the experiments (10 experiments) for multiple faults. For $c7552$, MSPI determines a minimum cardinality of 2 which is found by QBF [9], too. For $misex3$, MSPI finds the minimum number of fault candidates. LMBA also has a good accuracy with a reasonable time for multiple faults. For $pair$, LMBA determines a minimum cardinality of 3 while this method does not spend a long time. Also for gcd, LMBA has the best performance and accuracy.

Although the QBF [9] method has the best accuracy in the experiments, the heuristic methods often have a better performance for complex circuit structures and for fault candidates of large cardinality. For example for $pair$ and gcd in Table II, the QBF [9] approach is slow, while the heuristic approaches have a good performance and accuracy. Because the heuristic methods do not need to enumerate fault candidates explicitly. Especially an explicit enumeration for multiple faults reduces the performance of the exact approach, because the number of fault candidates increases exponentially with the fault cardinality [9].

Overall, the experimental results show that when we do not have a formal specification, LBA is a good method for the diagnosis of single faults while MSPI and LMBA are suitable for multiple faults. Thus, by selecting the best heuristics our flow is as accurate as an exact formal debugging in 71% of the experiments (24 experiments).

V. Conclusion

This paper introduced an approach for automating debugging procedures for designs using a testbench. Three techniques were proposed to generate diagnostic traces for deriving high quality counterexamples enhancing the diagnosis accuracy. LBA activates the local branches of each fault candidate. MSPI finds the sensitized paths including a minimum number of fault candidates. The advantages of both techniques are combined in LMBA. These techniques were evaluated and compared to random trace generation and one completely formal technique with respect to accuracy, run time and memory. The experimental results showed that our approach has an accuracy close to an exact formal debugging approach.

References

[1] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, “Fault diagnosis and logic debugging using boolean satisfiability,” *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[2] M. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, “Post-verification debugging of hierarchical designs,” in *Int’l Conf. on CAD*, 2005, pp. 871–876.

[3] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith, “A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test,” in *Int’l Conf. on CAD*, 2007, pp. 240–245.

[4] Y. Chen, S. Safarpour, J. M. Silva, and A. Veneris, “Automated design debugging with maximum satisfiability,” *IEEE Trans. on CAD*, vol. 29, no. 11, pp. 1804–1817, 2010.

[5] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[6] S. Safarpour and A. Veneris, “Abstraction and refinement techniques in automated design debugging,” in *Design, Automation and Test in Europe*, 2007, pp. 1182–1187.

[7] K. Chang, I. Markov, and V. Bertacco, “Fixing design errors with counterexamples and resynthesis,” in *ASP Design Automation Conf.*, 2007, pp. 944–949.

[8] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, “Increasing the accuracy of SAT-based debugging,” in *Design, Automation and Test in Europe*, 2009, pp. 1326–1332.

[9] A. Sülflow, G. Fey, and R. Drechsler, “Using QBF to increase accuracy of SAT-based debugging,” in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 641–644.

[10] K. Ravi and F. Somenzi, “Minimal assignments for bounded model checking,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 31–45.

[11] A. Groce and D. Kroening, “Making the most of BMC counterexamples,” *Electronic Notes in Theoretical Computer Science*, vol. 119, no. 2, pp. 67–81, 2005.

[12] M. N. Velev, “Comparison of schemes for encoding unobservability in translation to SAT,” in *ASP Design Automation Conf.*, 2005, pp. 1056–1059.

[13] S. Safarpour, A. Veneris, and R. Drechsler, “Improved SAT-based reachability analysis with observability don’t cares,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 5, pp. 1–25, 2008.

[14] T.-Y. Jiang, C.-N. Liu, and J.-Y. Jou, “Accurate rank ordering of error candidates for efficient HDL design debugging,” *IEEE Trans. on CAD*, vol. 28, no. 2, pp. 272–284, 2009.

[15] A. Veneris and I. N. Hajj, “Design error diagnosis and correction via test vector simulation,” *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.

[16] M. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M. Abadir, “Debugging sequential circuits using Boolean satisfiability,” in *Int’l Conf. on CAD*, 2004, pp. 204–209.

[17] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler, “WoLFram – a word level framework for formal verification,” in *IEEE/IFIP Int’l Symposium on Rapid System Prototyping (RSP)*, 2009, pp. 11–17.

[18] N. Eén and N. Sörensson, “An extensible SAT solver,” in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.