Debugging Sequential Circuits Using Boolean Satisfiability

Moayad Fahim Ali[1] Andreas Veneris[1,2] Sean Safarpour[1] Rolf Drechsler[3]
Alexander Smith[1] Magdy Abadir[4]

Abstract

Logic debugging of today's complex sequential circuits is an important problem. In this paper, a logic debugging methodology for multiple errors in sequential circuits with no state equivalence is developed. The proposed approach reduces the problem of debugging to an instance of Boolean Satisfiability. This formulation takes advantage of modern Boolean Satisfiability solvers that handle large circuits in a computationally efficient manner. An extensive suite of experiments with large sequential circuits confirm the robustness and efficiency of the proposed approach. The results further suggest that Boolean Satisfiability provides an effective platform for sequential logic debugging.

1 Introduction

As VLSI designs increase in size and complexity, errors become more frequent and harder to track. Common sources for these errors, also known as *design errors*, are bugs due to CAD tools and human interference [1][4]. Experience from a real life synthesis environment shows that the cardinality of these errors is usually small (2-3 errors) [4]. Given a design that fails verification, an engineer is often faced with the tedious task of identifying the source(s) of errors. With 60% of the overall VLSI design cost attributed to verification and debugging, it is evident that automated logic debugging tools are of great benefit.

Logic debugging is a challenging problem as the solution space grows exponentially with the increasing number of errors [12]. This is because the specification is usually treated as a "black box" controllable at the primary inputs and observable at the primary outputs. For example, the specification may be provided in a high-level language whereas the design is given in a logic-level implementation.

Debugging of combinational designs has an extensive literature and many efficient automated tools exist [4]. This is partly because this type of debugging bears similarity to fault diagnosis of full-scan designs which is a well examined topic [5]. On the other hand, there has been relatively little work in logic debugging for *sequential machines* [4]. This can be attributed to the increased complexity of the problem when no state equivalence information is available between the design and the specification [3] [6]. In this respect, one may compare this problem to the one of fault diagnosis for chips with no scan chains, a practically intractable problem [5]. Since most design blocks contain memory elements which are reshuffled many times during synthesis and optimization, debugging of sequential machines becomes a time-consuming and resource-intensive step in a VLSI design cycle with tight time-to-market constraints.

In this paper we propose a debugging method for *multiple* design errors in sequential circuits with no state equivalence information. The proposed method formulates the problem around the concept of Boolean Satisfiability (SAT). Therefore, it automatically benefits from recent advances in the field [2] [8] [9] [11] [14]. To the best of our knowledge, this is the first sequential circuit debugging method based on Boolean Satisfiability. Since logic debugging and fault diagnosis are similar in nature [12], the proposed approach applies to fault diagnosis of chips with no/partial scan chains.

An extensive suite of experiments in this paper demonstrates the robustness of the SAT-based logic debugging formulation. It is shown that large sequential circuits (over 10,000 gates) corrupted with multiple errors are handled in a few seconds. Since SAT can model a VLSI design at various degrees of abstraction, and due to the promising results presented here, this work encourages more research effort in SAT-based debugging.

The paper is organized in five sections. The next section contains background information and definitions. Section 3 contains the SAT-based sequential circuit debugging formulation and implementation. Section 4 reports experiments and Section 5 concludes this work.

2 Definitions

In this paper, we are interested in sequential circuits with primitive gates `AND`, `OR`, `NOT`, `NAND`, `NOR`, `XOR` and `XNOR` and fault-free memory elements (flip-flops). The *input* to the problem is a specification and an erroneous design initialized to a known state. The specification is given as a set of *sets of vectors* $V = V^{1,m_1}, V^{2,m_2}, \ldots, V^{k,m_k}$ with correct primary responses. Each element V^{j,m_j} of this set is a *test sequence* of input vectors $v^{j,1}, v^{j,2}, \ldots, v^{j,m_j}$ for m_j consecutive simulation *cycles*. In this sequence, the design gives correct primary output responses for all values of $m < m_j$ and an erroneous response for the last vector of the sequence, that is, vector v^{j,m_j}.

The set V can be obtained by random simulation and/or formal techniques. Test vector generation for counter-examples in sequential logic debugging is not the topic of this work [6]. Additionally, assuming that memory elements are fault-free allows both the specification and the netlist to reach a common initial state. For example, this state can be the reset state for all memory elements. Other advanced initialization procedures can be found in [6] as they are not the topic of this work. The *output* of the method is a set of potential error locations at which a correction(s) may be

[1]University of Toronto, Department of Electrical and Computer Engineering, Toronto, ON M5S 3G4 ({moayad, veneris, ssafarpo, smith}@eecg.toronto.edu)
[2]University of Toronto, Department of Computer Science, Toronto, ON M5S 3G4
[3]University of Bremen, Department of Computer Science, 28359 Bremen (drechsle@informatik.uni-bremen.de)
[4]Freescale Semiconductor, Inc., Austin, TX 78729 (m.abadir@freescale.com)

applied to rectify the design for the set of vectors V. These candidate locations can quickly provide the engineer with a small set of sites to concentrate on and perform correction. The output of the approach also provides useful information for the correction process. It is emphasized that error correction and formal verification techniques (following correction) are reviewed in [3] [4] [6] and are not dealt with here.

The proposed method uses a set of input test vectors to determine the source of the errors. Traditional simulation-based sequential logic debugging techniques use the set V of vectors to return sets of candidate error locations $E_1, E_2, \ldots, E_k$. Each set of candidate locations E_i is a set where the presence of some error explains the erroneous behavior for the respective input test sequence v^{i,m_i}. These sets are later *intersected* $E = E_1 \cap E_2 \cap \cdots \cap E_k$ to return the set E of error locations that some corrections on these lines rectify the circuit behavior for *all* test sequences. The proposed SAT-based method follows a similar approach.

We describe the algorithms on circuits with r primary inputs $X = x_1, x_2, \ldots, x_r$, initial state $Q_I = q_1, q_2, \ldots, q_s$ and t primary outputs $Y = (y_1, y_2, \ldots, y_t) = f(X, Q_I)$. We use $L = \{l_1, l_2, \ldots, l_n\}$ to represent internal circuit lines including stems and branches. The method in Section 3 adds new hardware which requires two extra lines for each original circuit line. We use the notation $S = \{s_1, s_2, \ldots, s_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ to label these lines.

In this presentation, variables for lines x_i, l_i, w_i and y_i are defined to model circuit constraints under simulation for *each* vector $v^{j,m}$, $m = 1, \ldots, m_j$ in sequence V^{j,m_j}. To avoid confusion, we use the notation $x_i^{j,m}, l_i^{j,m}, w_i^{j,m}$ and $y_i^{j,m}$ for these variables and $X^{j,m}, L^{j,m}$, $W^{j,m}$, and $Y^{j,m}$ for the respective sets (vectors) of variables. Under this notation, superscripts j and m match the indices of simulated test vector $v^{j,m}$ at cycle m. The notation $S = \{s_1, s_2, \ldots, s_n\}$ is used to indicate both SAT variables and line names. Variables for lines S are common to *all* test vectors j and *all* cycles m.

3 Debugging Sequential Circuits

Given a sequential logic netlist and a set of vectors V as defined in Section 2, the algorithm introduces new logic in the netlist to model error locations and error cardinality constraints. It then compiles this new circuit into a CNF formula Φ. This formula has *two components*.

The *first* component is the conjunction of $m_1 + \cdots + m_k$ CNF formulas $C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S)$ for all input test vector sequences $j = 1, \ldots, k$ and all simulation cycles $m = 1, \ldots, m_j$. Intuitively, each CNF $C^{j,m}$ enforces *constraints of sequence* V^{j,m_j} on the logic netlist and potential error sites through the inputs $X^{j,m}$ and the outputs $Y^{j,m}$. As will be explained, error locations are encoded in the circuit with extra hardware. The *second* component $E_N(S)$ encodes *constraints for the error cardinality* N. These constraints are also coded with new hardware. Since the number of errors N is unknown, its maximum value is a user-specified parameter. The algorithm starts with $N = 1$ and increments this value by 1 if the solver fails to return any location(s).

The complete formula Φ is expressed as:

$$\Phi \;=\; E_N(S) \cdot \prod_{j=1}^{k} \prod_{m=1}^{m_j} C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S)$$

In the subsections that follow, we describe how to compile each component of Φ.

3.1 Test Sequence Constraints

In this subsection we explain the first component of Φ, namely $\prod_{j=1}^{k} \prod_{m=1}^{m_j} C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S)$. To simplify this presentation, we develop the theory around an example that assumes a single input sequence V^{1,m_1}; that is, $k = 1$, with two cycles ($m_1 = 2$). At the end of the subsection, the results are generalized for multiple input sequences ($k > 1$) with an arbitrary number of simulation cycles.

When $k = 1$, the first component comprises of m_1 copies of CNF formula $C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S)$ representing the circuit. Each copy enforces different constraints on potential error locations and input/output behavior of the *correct* netlist. Clearly, this representation resembles the *Iterative Logic Array (ILA)* modeling of a sequential netlist in test generation [5]. In the ILA representation, a sequential circuit is "unrolled" in time. This is performed using identical copies of its combinational circuitry at different simulation cycles where the output of the memory elements from cycle i is connected to the appropriate gate primitives in cycle $i+1$. For example, the ILA representation of the sequential circuit in Fig. 1(a) is shown in Fig. 1(c) for some input test sequence two cycles long. The equivalence between these two representations becomes evident if we draw one "time-slice" of the circuit of Fig. 1(a) as shown in Fig. 1(b).

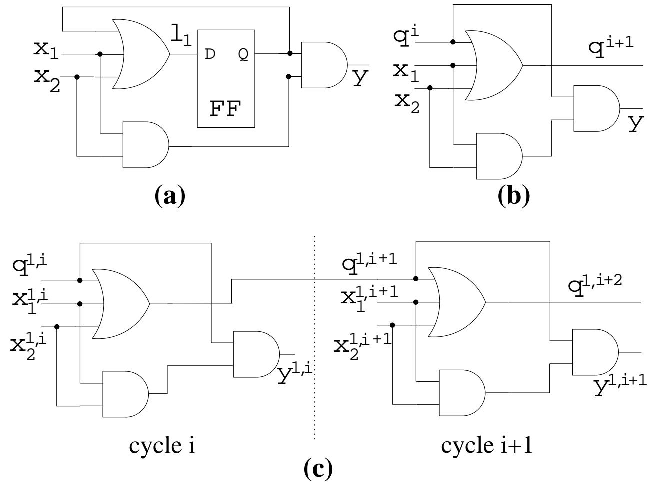

Figure 1: Example circuit

Given an erroneous netlist and a test sequence with m_1 cycles, the circuit is first transformed to its ILA representation. Error locations are then modeled by attaching extra hardware. This hardware reflects the potential of an error on lines of the circuit for *all* m_1 simulation cycles. It should be noted that this hardware does not require the error to be excited in all cycles but it merely indicates the presence of an error that may or may not be excited. In particular, to model the presence of a fault on line $l_i^{1,m}$, a multiplexer with select line s is attached to every instance $m = 1, \ldots, m_1$ of this line for the different cycles. All these m_1 multiplexers are later translated into CNF for Φ. The first input of each multiplexer is attached to the line $l_i^{1,m}$ and the second input is attached to a new line $w_i^{1,m}$. The output of each multiplexer is connected to the gate where $l_i^{1,m}$ was originally connected. It is important to note that all m_1 multiplexer copies share the *same* select line s.

Consider again the circuit in Fig. 1(a) and assume the design error to be a gate replacement of the `OR` gate that drives

line l_1 to a NOR gate. Since the gate is an input only to a memory element, any input test sequence needs at least two cycles to detect the error at the primary output [5]. The reader can verify that vector sequence $V^{1,2} = \{v^{1,1}, v^{1,2}\} = \{x_1^{1,1}x_2^{1,1}, x_1^{1,2}x_2^{1,2}\} = \{10, 11\}$ is such a sequence that produces an erroneous primary output value 0 if the initial state is 0.

The presence of an error on line l_1 can be represented with two multiplexers with common select line s at respective positions $l_1^{1,1}$ and $l_1^{1,2}$ of the ILA representation of this circuit, as shown in Fig. 2. The first input of each multiplexer is connected to the output of the respective NOR gate and the second input is connected to a new line $w^{1,i}$, to model the potential error in the respective cycle i. The output of the multiplexer is connected to the original output of the NOR gate. Observe that the functionality of the faulty circuit is selected when the value of the common select line $s = 0$. On the other hand, a new circuit with "free" lines $w^{1,1}$ and $w^{1,2}$ is selected when $s = 1$.

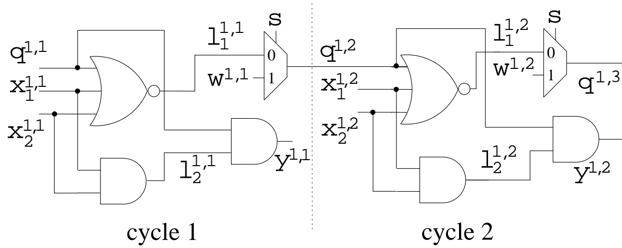

Figure 2: Debugging in two cycles

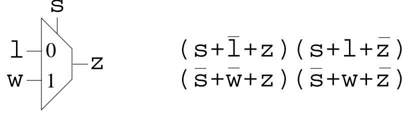

Figure 3: Multiplexer implementation

As shown in Fig. 3, the CNF for a multiplexer requires only four clauses. As a result, the CNF for the ILA circuit implementation for cycle i is $\mathcal{F}^i = (\overline{q}^{1,i} + \overline{l}_1^{1,i}) \cdot (\overline{x}_1^{1,i} + \overline{l}_1^{1,i}) \cdot (\overline{x}_2^{1,i}+\overline{l}_1^{1,i}) \cdot (q^{1,i}+x_1^{1,i}+x_2^{1,i}+l_1^{1,i}) \cdot (x_1^{1,i}+\overline{l}_2^{1,i}) \cdot (x_2^{1,i}+\overline{l}_2^{1,i}) \cdot (\overline{x}_1^{1,i}+\overline{x}_2^{1,i}+l_2^{1,i}) \cdot (s+\overline{l}_1^{1,i}+q^{1,i+1}) \cdot (s+l_1^{1,i}+\overline{q}^{1,i+1}) \cdot (\overline{s}+w_1^{1,i}+\overline{q}^{1,i+1}) \cdot (\overline{s}+\overline{w}_1^{1,i}+q^{1,i+1}) \cdot (q^{1,i}+\overline{y}^{1,i}) \cdot (l_2^{1,i}+\overline{y}^{1,i}) \cdot (\overline{q}^{1,i}+\overline{l}_2^{1,i}+y^{1,i})$. Hence, the CNF for the circuit in Fig. 2 is $\mathcal{F} = \mathcal{F}^1 \cdot \mathcal{F}^2$.

Once multiplexers are introduced at *every* circuit line for all cycles, the updated ILA circuit representation is translated into CNF. To get the final $C^{j,m}$, we need to insert clauses to represent input/output specification constraints for the erroneous circuit and *all* m_1 cycles of test sequence V^{1,m_1}. This is done (for every cycle) with a set of unit-literal clauses for primary input variables $x_1^{1,m}, x_2^{1,m}, \ldots, x_r^{1,m}$, correct primary output variables $y_1^{1,m}, y_2^{1,m}, \ldots, y_t^{1,m}$, and initial state variables Q_I. These literals agree with the respective logic values of the vector $v^{j,m}$ and circuit response at cycle m; that is, if $v^{j,m}$ assigns a logic 1 (0) to input x_l then $x_l^{j,m}$ ($\overline{x}_l^{j,m}$) appears as a unit-literal clause in the formula.

Example 1: Recall the circuit from Fig. 1 in which the gate replacement error is detected in the second cycle with test sequence $V^{1,2} = \{10, 11\}$ because $y_{err}^{1,2} = 0$ and $y_{corr}^{1,2} = 1$. The ILA representation of the erroneous circuit for $V^{1,2}$ is shown in Fig. 2. To enforce the correct input/output vector constraints from $V^{1,2}$ on the ILA representation, we add unit-literal clauses $\overline{q}^{1,1}$, $x_1^{1,1}$, $\overline{x}_2^{1,1}$, $\overline{y}^{1,1}$ $x_1^{1,2}$, $x_2^{1,2}$ and $y^{1,2}$. Unit-literal clause $\overline{q}^{1,1}$ is added because we assume that the memory elements of the circuit can be correctly initialized to their reset (0) state. Therefore, the final CNF formula for $V^{1,2}$ is $\mathcal{F}' = \prod_{m=1}^{m_j} C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S) = \mathcal{F} \cdot \overline{q}^{1,1} \cdot x_1^{1,1} \cdot \overline{x}_2^{1,1} \cdot \overline{y}^{1,1} \cdot x_1^{1,2} \cdot x_2^{1,2} \cdot y^{1,2}$. Observe that if $\mathcal{F}'$ is passed to a SAT solver, the engine will *necessarily* assign $s = 1$. The assignment $s = 0$ will cause the solver to backtrack with a conflict because the erroneous circuit is requested to produce a correct primary output behavior.

This process is repeated for every test sequence V^{j,m_j}, $j = 1 \ldots k$ to get CNFs $C^{j,m}(L^{j,m}, W^{j,m}, X^{j,m}, Q_I, Y^{j,m}, S)$, the product of which forms the first component of Φ. Note that each such formula requires a new set of variables for primary inputs ($X^{j,m}$), primary outputs ($Y^{j,m}$), internal circuit lines ($L^{j,m}$), error lines ($W^{j,m}$) and initial state Q_I. This is because every input test vector may translate into a different set of constraints for these variables. However, only one set of select line variables $S = s_1, s_2, \ldots, s_n$ is required because the error locations of a solution must satisfy *all* vector constraints simultaneously. The second component of Φ, described next, constrains the cardinality N of error lines.

3.2 Error Cardinality Constraints

The second component $E_N(S)$ enforces appropriate constraints in Φ that require a solution with at most N error locations. This component also requires extra hardware to be added to the enhanced ILA form from Subsection 3.1. When the updated ILA is translated to CNF, we obtain Φ. The following example gives the intuition for $E_N(S)$ when $N = 1$. Following this example, we present the hardware construction that generalizes the idea for multiple errors.

Example 2: Consider formula $\mathcal{F}'$ as computed in Example 1. This formula models the erroneous circuit under simulation of test sequence $V^{1,2}$. Assume multiplexer select line s is introduced in $\mathcal{F}'$ as a unit-literal clause to produce formula $\mathcal{F}'' = \mathcal{F}' \cdot s$. Given $\mathcal{F}''$, a SAT solver assigns $s = 1$ and attempts to find a satisfying assignment for the circuit lines and the "free" variables $w^{1,1}$ and $w^{1,2}$ so that the circuit emulates the specification for test sequence $V^{1,2}$. Note that once the solver returns successfully, the logic value assignments for these free variables are those found in the specification on lines $l_1^{1,1}$ and $l_1^{1,2}$, respectively (if such lines exist).

The general idea for $E_N(S)$ is an extension of the example above. That is, formula Φ can be updated with clauses that enumerate exhaustively all possible sets of error sites. These clauses will enforce subsets $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ of S to be set to a logic 1 and indicate that N error(s) are activated. Although this formulation is intuitive, it requires an exponential number of clauses to be inserted *explicitly* in Φ.

To overcome a memory explosion with increasing values of N, a different approach is taken with an encoding of $E_N(S)$ using the hardware construction as shown in Fig. 4(a). In this figure, thick lines indicate buses of $O(logn)$ bit-width ($N \leq n$) and all other lines represent single bit lines. This hardware acts as a *counter* forcing the SAT solver to "enumerate" sets of N fault sites. It performs a bitwise addition of the multiplexer select lines $S = s_1, s_2, \ldots, s_n$ and compares the result to the user-defined number of faults N. The output of the comparator is "forced" to logic 1 with a unit-literal clause so that the bitwise addition of the members of S (that is, the set of fault sites enumerated) is always equal

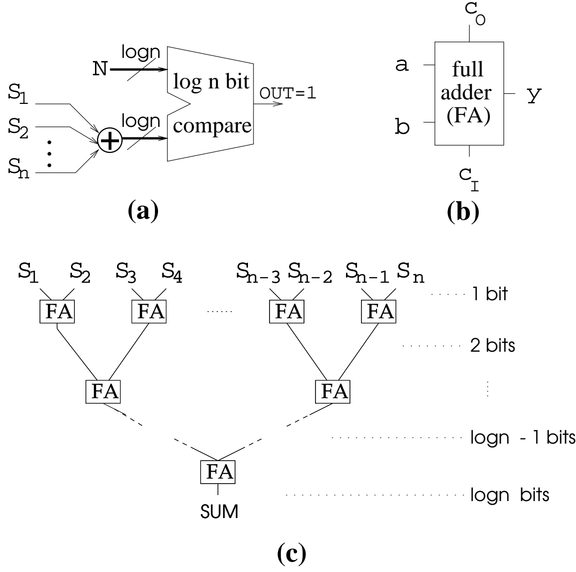

Figure 4: Counter implementation

to N. As with the select lines, the variables introduced for this hardware are common to all test sequences V.

Intuitively, this *implicit* hardware representation for $E_N(S)$ provides a trade-off between time and space. Experiments show that modern SAT solvers take advantage of this trade-off; they avoid an exponential explosion in the time domain while their memory requirements remain low. In the remainder of this section, we show how to construct the counter hardware in CNF with $O(n)$ number of clauses.

As seen in Fig. 4(a), the counter contains an *adder* for the select lines and a *comparator*. Assume that the binary representation of the integer passed from the adder to the comparator is $b_{\log n-1} \ldots b_1 b_0$. A comparator for SAT-based debugging of 2 faults is formed by adding CNF $\overline{b}_{\log n-1} \cdot \overline{b}_{\log n-2} \cdot \ldots \cdot \overline{b}_2 \cdot b_1 \cdot \overline{b}_0$ in $E_N(S)$. This ensures that two select lines are always 1 and all others are forced to 0, otherwise Φ is not satisfiable. In a similar manner, a comparator can be constructed in CNF for any value of N with $\log n$ extra unit-literal clauses.

An implementation for the adder with $O(n)$ clauses is found in Fig. 4(c). The 1-bit values of the select lines are added progressively in a binary tree fashion to compute the $\log n$ bit sum. The binary tree has $\log n$ levels, with select lines at level 0. At each level $i = 0 \ldots \log n$, $2^{\log n - i}$ integers are added pairwise. Each such integer is $i+1$ bits long, and integer bits are added with a sequence of full-adders shown in Fig. 4(b). A full-adder can be encoded in CNF using 14 clauses (6 clauses if the carry-in is omitted). The CNF size for the adder is proportional to the number of CNF variables (bits) used to hold the values of the select lines and all intermediate results of the adder tree. Hence, the total number of these CNF variables is at most:

$$
\begin{aligned}
\#\ CNF\ variables &\leq \sum_{i=0}^{\log n} 2^{\log n - i}(i+1) \\
&= 2^{\log n}\left[\sum_{i=0}^{\log n} i\left(\frac{1}{2}\right)^i + \sum_{i=0}^{\log n}\left(\frac{1}{2}\right)^i\right] \\
&\leq 2^{\log n}\left[\sum_{i=0}^{\infty} i\left(\frac{1}{2}\right)^i + \sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i\right] \\
&= 4 \cdot 2^{\log n} \\
&= O(n)
\end{aligned}
$$

The calculation uses the fact that $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ and $\sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}$ when $|x| < 1$. Since the full-adders contribute a constant multiplicative factor of clauses, we conclude that the number of clauses for the counter is $O(n)$.

3.3 Implementation

In this section we discuss memory requirements and run-time heuristics. From the previous discussion, it is clear that the *space requirements* for Φ are *linear* $O(nkm)$ in the number of circuit lines n, the number of test sequences k and the length m of these sequences.

Although space efficient, for large industrial circuits the formula Φ may grow quickly with the number of vectors. To keep the space requirements low yet preserve efficiency, we compile a set of formulas $\Phi_1, \Phi_2 \ldots, \Phi_{\lceil \frac{k}{p} \rceil}$. Each formula encodes constraints for p distinct test vectors and it needs only $O(npm)$ space where $p < k$. In creating Φ_i, we only place multiplexers on fault sites at which solutions to Φ_{i-1} are found. Intuitively, Φ is the conjunction of all these formulas Φ_i. The rationale of the heuristic lies in the fact that in diagnosis a small number of vectors usually screens the majority of invalid candidates [4] [12]. Consequently, only a few fault sites and respective multiplexers (in the experiments less than 5% of the circuit lines on average) are introduced in subsequent phases of the algorithm. The benefit of this heuristic is further examined during experiments.

To improve performance further, the algorithm runs in *two passes* as it originally inserts multiplexers only at structural circuit *dominators* [5]. Once a set of dominator-solutions is identified, a second pass is run to find solutions in their respective fan-in cones. Running the method in this two-pass fashion keeps the size of the added hardware and the solution space small, which makes it easier for the solver.

Since the CNF of the circuit presented to the SAT solver is replicated for a number of cycles for each input/output vector sequence, the SAT instance may become large. To ease the task of the SAT solver, test sequences are sorted in increasing size $m_{i_1} \leq m_{i_2} \leq \cdots \leq m_{i_k}$ and presented in this order to the SAT solver. This heuristic ensures that the first few SAT instances (which tend to be the hardest ones) Φ_1, $\Phi_2, \ldots$ are solved first. These instances have a relatively small size, and so they present an easier task to the SAT solver. Larger sequences are solved later when the process has already identified a set of error locations, a fact that eases the task of the solver.

Given an erroneous design, there may be many ways one can resynthesize and correct it [4][5][13]. The existence of many candidate correction locations (for a fixed error(s)) provides additional flexibility to the design engineer during debugging. It is also an important fact for logic optimization techniques that use logic debugging as their underlying engine [13].

The single-solution SAT-based logic debugging method presented in Section 3 can be easily modified to an *all-solution* engine as follows. As soon as a solution with error sites $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ is returned, the clause $(s'_{i_1} + s'_{i_2} + \cdots + s'_{i_N})$ is immediately added as a *learned clause*. This causes the solver to backtrack and search for other error sites in the remaining solution space reusing part of the existing

Table 1: Sequential debugging for single errors

ckt name	# of gates	# cycles min.	# cycles max.	# loc. dom.	# loc. all	CPU (sec) dom.	CPU (sec) all
s298	142	2	4	3	**6**	0.02	**0.02**
s444	171	2	3	2	**6**	0.05	**0.02**
s1196	479	1	3	1	**4**	0.21	**0.07**
s1488	522	1	5	2	**10**	0.22	**0.06**
s1494	531	1	3	2	**10**	0.22	**0.05**
s3384	1,610	2	6	3	**7**	1.97	**0.95**
s4863	1,817	2	6	4	**9**	1.05	**0.53**
s5378	1,407	1	2	2	**6**	0.56	**0.23**
s6669	2,908	1	6	10	**25**	1.30	**0.60**
s9234	1,185	3	3	7	**12**	0.48	**0.41**
s15850	4,303	1	4	11	**32**	0.72	**0.30**
s35932	11,186	1	6	5	**7**	2.74	**2.43**
s38417	13,940	3	4	3	**12**	6.52	**2.44**
b12	932	2	6	21	**30**	0.25	**0.22**
b14	5,924	2	5	4	**8**	5.10	**2.98**
b15	8,027	4	6	24	**34**	3.72	**0.82**
b21	13,169	3	6	4	**11**	36.90	**14.30**
Avg.	4,014	1.88	4.59	6.35	**13.40**	3.65	**1.55**

Table 2: Sequential debugging for double errors

ckt name	# of clauses	# cycles min.	# cycles max.	# loc. dom.	# loc. all	CPU (sec) dom.	CPU (sec) all
s298	10,166	1	2	13	**29**	0.04	**0.02**
s444	10,677	3	3	6	**31**	0.04	**0.01**
s1196	18,677	1	3	5	**44**	0.25	**0.03**
s1488	24,753	1	2	3	**33**	0.67	**0.07**
s1494	29,782	1	3	3	**42**	1.36	**0.10**
s3384	162,771	6	6	9	**42**	17.60	**3.82**
s4863	111,909	2	5	5	**28**	33.20	**5.98**
s5378	62,727	1	5	14	**43**	1.56	**0.52**
s6669	217,768	2	5	13	**33**	36.20	**12.70**
s9234	86,114	2	4	6	**35**	4.18	**0.77**
s15850	269,630	1	4	42	**164**	22.80	**5.92**
s35932	719,360	2	3	16	**42**	95.80	**38.50**
s38417	1,066,607	1	2	10	**25**	1.55	**0.74**
b12	111,359	2	2	8	**29**	0.16	**0.05**
b14	580,917	2	2	6	**28**	0.81	**0.29**
b15	902,535	2	2	4	**25**	1.82	**0.39**
b21	1,201,102	1	1	9	**41**	1.28	**0.36**
Avg.	328,638	1.82	3.17	10.10	**42.00**	12.90	**4.13**

computation. Details about this heuristic, as well as other heuristics that improve performance, are found in [12].

Finally, the proposed method has the added advantage of providing information that is useful in correction. To see this, the presented formulation does not make any assumptions on the logic value of the error for each test vector. Given sets of logic assignments $w_{i_1}^{j,m}, w_{i_2}^{j,m}, \ldots, w_{i_N}^{j,m}$ for values of j and m on the "free" lines for respective candidate circuit lines $l_{i_1}, l_{i_2}, \ldots, l_{i_N}$ returned by the algorithm, these assignments are required on these lines to guarantee that the netlist emulates the specification for all test sequences. These logic assignments can be used by the test engineer to derive corrections as in traditional logic debugging [4]. Due to all these characteristics, we conclude that Boolean Satisfiability provides an attractive platform for sequential logic debugging and fault diagnosis.

4 Experiments

The automated sequential circuit debugger described in the previous sections is implemented in C++ using zChaff [9] as the underlying satisfiability engine. The experiments are conducted on a Pentium IV 2.8GHz Linux platform with 2GB of memory using ISCAS'89 and ITC'99 circuits optimized using SIS (`script.rugged`) [10]. Errors are inserted in the correct circuit and simulated to obtain the set of vector sequences with failing responses. Each experiment uses 20 test sequences. All errors are a random functional change to random circuit lines excluding primary input/output lines. Each experimental result shown here is an average of ten experiments and all run-times are in seconds.

Tables 1 and 2 show results in a similar manner for single error and double error experiments, respectively, except for the second column. Column one contains the circuit name. Column two of Table 1 has the number of gates and that of Table 2 shows the maximum number of CNF clauses for any Φ_i for both single and double errors. It can be seen that the SAT formula remains reasonably small even for large circuits. This confirms the potential of the heuristics in subsection 3.3.

Columns three and four present the minimum and maximum number of cycles required to observe the error(s). These numbers represent the range of the m_j values discussed earlier. Column five shows the number of error locations found for the first pass of structural dominators (Section 3.3). The next column shows the number of error locations upon termination of the algorithm (second pass). It is seen that the method exhibits very good resolution; the number of locations is small enough to aid the task of the verification engineer in locating the source of error(s).

The run time (per location) to return the error sites from the first pass is shown in column seven. The total time for the first pass is found if we multiply the number of dominator locations (column five) with the one in column seven. The total run time (per location) for the entire circuit is shown in column eight. Parameter p (subsection 3.3) is set to a constant value of 5 for all the experiments shown here. The times reported in the tables confirm that the method offers excellent resolution in a computationally efficient manner.

As shown in the previous section, the space requirement of the proposed method is $O(npm)$, where n is the number of circuit lines, p the number of test sequences in Φ_i and m the maximum length of these test sequences. To further analyze the behavior of the method, we show results where one of the parameters n, p or m changes while the other two remain constant during debugging for single errors.

Fig. 5 illustrates the relationship between the circuit size n and the overall run time per solution when the value of m remains constant. The graph shows that the method scales linearly with the circuit size. This indicates that SAT provides an efficient platform for sequential logic debugging of large industrial designs.

Fig. 6 illustrates the relationship between the parameter p and the overall CPU time when m is constant. Three sample circuits of different size suggest that the best (performance wise) value for p is 5. This is because as p grows, so does the size of the CNF formula, which makes the SAT instance harder to solve. On the other hand, smaller values of p enforce less tight constraints and increase the number of potential locations the SAT solver returns. The efficiency achieved with $p = 5$ balances these two parameters.

The analysis for varying values of m when $p = 5$ is found in Fig. 7. Similarly to Fig. 5, the CPU time is found to scale well with an increasing number of cycles. This similarity between the two behaviors is partly due to the fact that both m and n are directly associated with the size of the

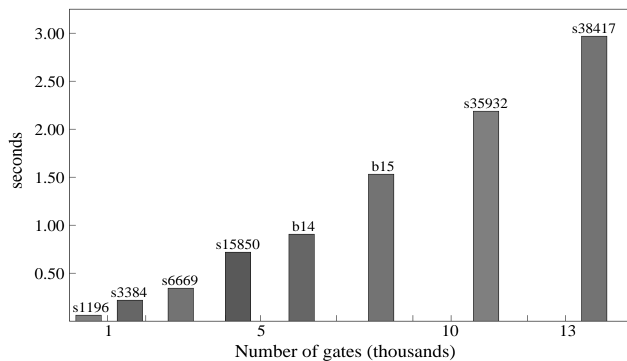

Figure 5: Run time vs. circuit size n

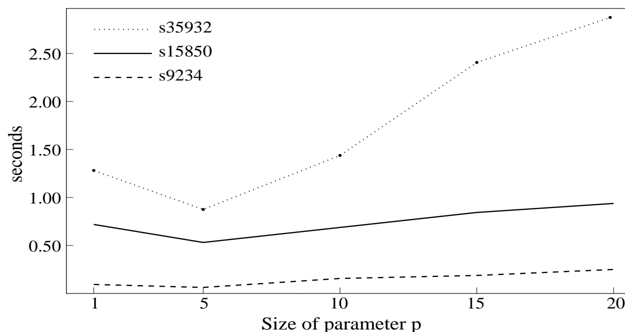

Figure 6: Run time vs. Φ_i size p

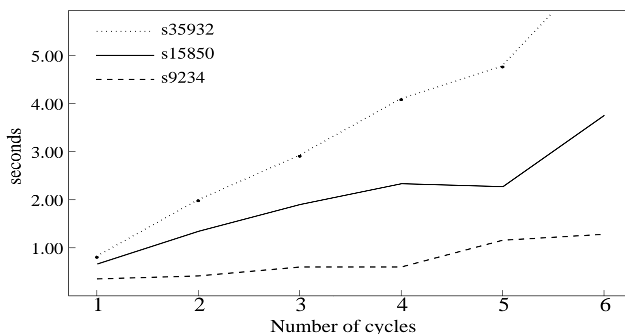

Figure 7: Run time vs. number of cycles m

CNF formula Φ_i. As the CNF formula increases, so does the time to solve the overall problem.

Table 3 provides insight into the behavior of the underlying SAT solver during this SAT-based debugging formulation for single errors. It is interesting to see that the number of backtracks for the experiments shown in Table 1 is quite small. This in turn means that the SAT solver makes few "wrong" decisions that lead to conflicts and backtracks.

This behavior is due to the sequential debugging SAT-based instance, as formulated herein, being a problem in which solution constraints are tightly specified in terms of the circuit structure and input test sequence(s). Therefore, the majority of the circuit lines acquire their "correct" logic values through Boolean Constraint Propagation [9]. Hence, we may conclude that the solver is given a relatively easy problem to solve irrespective of the circuit size. These conclusions reinforce the fact that Boolean Satisfiability is an efficient, practical and robust way to perform sequential logic debugging in industrial designs.

Table 3: Number of backtracks

ckt	# backtracks	# loc.	ckt	# backtracks	# loc.
s444	18	6	s5378	27	6
s9234	47	12	s15850	108	32
s35932	214	7	s38417	55	12

5 Conclusion

A sequential debugging technique for multiple design errors using Boolean Satisfiability was presented for circuits with no state equivalence. The method efficiently translates the problem of sequential debugging into a Boolean Satisfiability instance. Therefore, it automatically benefits from advances to SAT solvers to increase the efficiency of the solution. As demonstrated through experiments, the proposed approach performs very well for large circuits corrupted with multiple errors. Theory and experiments confirm its practicality and encourage further research effort towards novel SAT-based debugging techniques.

References

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation", *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 1988.

[2] D. Chai and A. Kuehlmann, "Fast pseudo-Boolean constraint solver", *Proc of DAC*, pp. 830-835, 2003.

[3] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms Chapter 8*, Kluwer Academic Publishers, 2000

[4] S. Y. Huang and K. T. Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[5] N. Jha and S. Gupta, *Testing of Digital Systems*, Cambridge University Press, 2003.

[6] T. Kropf, *Introduction to Formal Hardware Verification, Chapter 3*, Springer-Verlag, 1999.

[7] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," in *IEEE Trans. on CAD, vol. 11, no. 1, pp. 4-15,* Jan. 1992.

[8] F. Lu, L.-C. Wang, K.-T. Cheng and R. Y.-Y. Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning," in *Proc. of IEEE DATE, pp. 892-897,* 2003.

[9] M.H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of DAC, pp. 530-535,* 2001.

[10] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of ICCAD, pp. 328-333,* 1992.

[11] J. P. M.-Silva and K. A. Sakallah, "GRASP – A Search Algorithm for Propositional Satisfiability," in *IEEE Trans. on Computers, vol. 48, no. 5, pp. 506-521,* May 1999.

[12] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability," in *Proc. of ASP-DAC, pp. 218-223,* 2004.

[13] A. Veneris and M. S. Abadir, "Design Rewiring Using ATPG," in *Proc. IEEE Trans. on Computer-Aided Design, vol. 21, no. 12, pp. 1469-1479,* Dec. 2002.

[14] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee, "Managing Don't Cares in Boolean Satisfiability," in *Proc. of IEEE DATE, pp. 260-265,* Feb. 2004.