Automated Design Debugging with Maximum Satisfiability

Yibin Chen, *Student Member, IEEE*, Sean Safarpour, *Member, IEEE*,
Joao Marques-Silva, *Senior Member, IEEE*, Andreas Veneris, *Senior Member, IEEE*

***Abstract*— As contemporary VLSI designs grow in complexity, design debugging has rapidly established itself as one of the largest bottlenecks in the design cycle today. Automated debug solutions such as those based on Boolean Satisfiability (SAT) enable engineers to reduce the debug effort by localizing possible error sources in the design. Unfortunately, adaptation of these techniques to industrial designs is still limited by the performance and capacity of the underlying engines. This paper presents a novel formulation of the debugging problem using MaxSat to improve the performance and applicability of automated debuggers. Our technique not only identifies errors in the design but also indicates when the bug is excited in the error trace. MaxSat allows for a simpler formulation of the debugging problem, reducing the problem size by 80% compared to a conventional SAT-based technique. Empirical results demonstrate the effectiveness of the proposed formulation as run-time improvements of $4.5\times$ are observed on average. This work introduces two performance improvements to further reduce the time required to find all error sources within the design by an order of magnitude.**

I. Introduction

The complexity of digital designs in the semiconductor industry has increased exponentially over the last few decades. Even though innovations in design tools and methodologies have made significant progress in reducing human error, designing a circuit free of functional errors remains a challenge. Functional verification and debugging tasks have established themselves as major bottlenecks in the design process, requiring up to 70% of the total design effort [1].

Due to the risk inherent in today's highly complex design flows, engineers are increasingly adopting automated design and verification tools to ensure correctness [2], [3]. However, once verification fails, localizing and rectifying the erroneous behavior (debugging) remains a predominantly manual task lacking sufficient automation. As design complexity increases, larger design and verification teams and the usage of third party IP further complicate the debugging task. It is therefore not surprising that on average 60% of the verification effort is spent on debugging [4] and the costs of identifying the root cause of an error are growing rapidly. Automated debugging solutions are essential to accelerate these tasks.

Techniques based on simulation [5], path tracing [6], and Binary Decision Diagrams (BDDs) [7] have been proposed in literature to enhance the efficiency of error localization and diagnosis. More recently, the performance of engines based on Boolean Satisfiability (SAT) [8] have encouraged further research into formal debugging techniques. The purpose of these techniques is to automatically identify the source of functional errors based on constraints specifying the circuit's expected behavior.

In the SAT-based debugging approach presented in [8], design errors are located by first adding correction models to the circuit implementation. A correction model can be dedicated hardware that indicates whether a circuit component can be rectified to correct the circuit. The problem is then transformed into a Boolean formula in Conjunctive Normal Form (CNF) and constrained using the expected behavior of the circuit. Solving the CNF problem using a Boolean SAT solver implicates a set of suspect error locations which could be responsible for the error.

Based on the above formulation numerous derivative works such as [9], [10] and [11] have significantly improved the performance and scalability of automated debuggers to make them applicable for industrial designs. However, enhancing the performance (run-time) and capacity (memory) of these techniques remain an ongoing challenge. Contemporary debuggers still have difficulty handling large industrial designs and the correction models greatly increase the size of the CNF problem. Moreover, existing techniques are limited to identifying where the most likely error locations are in the design (*spatial* error locations). They do not determine when in the error trace the bug occurs (*temporal* error locations). Temporal information is very important for diagnosing and correcting the design [12] as they allow the designer to track down more easily when in the error trace the error occurs.

In this paper a novel approach to design debugging using Partial Maximum Satisfiability (Partial MaxSat) is presented [13], [14]. Similar to SAT problems, Partial MaxSat problems are Boolean Satisfiability problems expressed in CNF. However, instead of finding an assignment to a satisfiable CNF formula, MaxSat solvers find the largest satisfiable subset of CNF clauses in an unsatisfiable problem. In essence, the inverse of that subset identifies a set of error locations which could be responsible for the bug. Our novel formulation

Yibin Chen is with Vennsa Technologies, Inc., Toronto, ON M5V 3B1, Canada (email: chenyibin@gmail.com).

Sean Safarpour is with Vennsa Technologies, Inc., Toronto, ON M5V 3B1, Canada (email: sean@vennsa.com).

Joao Marques-Silva is with the School of Electronics and Computer Science, University College Dublin, Belfield, Dublin 4, Ireland (email: jpms@ucd.ie).

Andreas Veneris is with the Department of Computer Science and the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (email: veneris@eecg.toronto.edu).

Manuscript received September 27, 2009; revised March 30, 2010. This paper was recommended by Associate Editor Robert F. Damiano.

allows us to identify all errors locations both *spatially* and *temporally*, while significantly reducing the size of CNF problems compared to SAT-based debugging. Our Partial MaxSat formulation can improve the performance of debugging while providing solutions at a finer granularity.

In addition, this paper also proposes two techniques to improve debugger performance. First, we present a method to quickly enumerate similar Partial MaxSat solutions based on results obtained from the solver. These allow us to reduce the number of solver iterations required to find all solutions.

Next, we show how tuples of related clauses can be grouped such that they are treated as a single high-level constraint. These tuples increase the granularity of the solutions and reduce the search space of the problem, improving run time. Groupings based on modules and time frames allow for more relevant solutions to be found at lower error cardinalities. We demonstrate how groupings can be used to implement hierarchical debugging [15] with Partial MaxSat to find both module-level and gate-level error locations.

Experimental results show that our approach can find error locations $4.5\times$ faster on average and using 80% smaller CNF problems. The optimization techniques presented allow us to identify all MaxSat solutions using 87.5% fewer solver iterations while improving the run time per iteration by $1.6\times$. As the performance of MaxSat solvers is still improving at a rapid pace [16], our formulation will become more effective as engine technology progresses.

The remainder of this paper is structured as follows. First, some background information on SAT-based automated debugging and MaxSat solvers is provided. Then our MaxSat formulation for combinational circuits is presented in Section III. The formulation is extended to finding spatial and temporal error locations in sequential circuits in Section IV. An algorithm to reduce the number of MaxSat iterations to find all solutions is presented in Section V. Section VI uses groupings to leverage the hierarchical structure of the circuit to improve performance. Finally the experiments, related works, and conclusions are provided in Sections VII, VII and IX respectively.

II. Background

A. SAT-based Automated Debugging

Functional design debugging occurs at the stage of the design cycle when the Register Transfer Level (RTL) implementation of the design has failed verification (simulation or formal). The output of the verification effort is an error trace proving the existence of a bug. In the context of this paper, design debugging seeks to locate all possible error sources in the implementation.

In the SAT-based debugging formulation presented in [8], the circuit is first enhanced with a correction model by adding a multiplexer at the output of each gate or module. The select line of the multiplexer controls whether the correction model is active or inactive. If the correction model is inactive, the circuit behaves according to its implementation. If the correction model is active, the output of the gate is left unconstrained and can be replaced with a value that can correct the error.

For sequential circuits, an Iterative Logic Array (ILA) [6] is constructed by unrolling the circuit for the length of the error trace. The correction model is added as in the combinational case but multiplexer select lines for the same gate are shared across all time frames. The problem is converted into CNF and constrained using the the stimulus vector and the expected output values from the error trace. Additional constraints (usually implemented using an adder) limit the number of correction models that can be simultaneously activated. This number is also known as the error cardinality N_g of the problem.

Finding all satisfying assignment to the resulting formula effectively finds a set of functionally equivalent error locations, that if corrected could fix the bug. Sets of error locations are said to be *functionally equivalent* if they cannot be functionally distinguished from each other under a given stimulus trace [6]. The debugger is limited to finding the set of all functionally equivalent error locations.

Based on this formulation numerous advances to enhance the performance of debuggers have been proposed. The work from [9] presents a QBF based debugging formulation using universal quantifiers which allows for sequential circuit debugging without the need for an ILA. In [11] the concept of abstraction refinement is used to reduce the size of the SAT problem and improve performance. In [10], the authors take advantage of unsatisfiable cores to speed up the debugging process for multiple fault diagnosis problems. This approach extracts a set of unsatisfiable cores from the CNF problem and prunes potential error locations not contained in any of the cores. A SAT-based exact debugger then finds the error locations from the reduced problem.

B. Maximum Satisfiability

This section reviews MaxSat [16] and its extensions, and briefly overviews recent algorithms for MaxSat, capable of handling large complex problem instances. Given an *unsatisfiable* CNF formula Φ, the MaxSat problem consists of identifying an assignment to the problem variables such that the number of satisfied clauses from Φ is maximized [17]. The MaxSat problem is a well-known NP-Hard optimization problem.

In the *Partial MaxSat* [18] problem the CNF formula is organized into a set of *hard* clauses, which must be satisfied, and a set of *soft* clauses, which may or may not be satisfied, *i.e.* $\Phi = \Phi_H \cdot \Phi_S$. For Partial MaxSat problems the objective is to find an assignment that satisfies all the hard clauses and that maximizes the number of satisfied soft clauses.

In the remainder of this paper, *hard* clauses will be represented in square brackets and *soft* clauses in round brackets. For example, consider the following formula:

$$\Phi = [x_1 + \overline{x_2}][x_3] \cdot (\overline{x_1})(x_2)(\overline{x_3} + x_1) \tag{1}$$

The first two clauses are hard clauses, and so must be satisfied, whereas the remaining three clauses are soft clauses and may or may not be satisfied.

In the recent past [17], the most effective MaxSat algorithms have been based on branch-and-bound (B&B), supported by

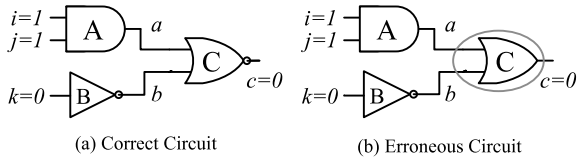


Fig. 1. Correct and erroneous combinational circuit

effective lower bounding and dedicated inference techniques. Nevertheless, most of the experimental evaluation associated with B&B MaxSat solvers assume random and handmade problem instances, which unfortunately often bear little relationship with hard industrial instances. Recent work has addressed alternative approaches, aiming the use of MaxSat algorithms in industrial settings, and focusing on instances derived from realistic applications. The most effective algorithms are based on solving MaxSat with unsatisfiable sub-formula identification and relaxation [18]–[20].

III. Clause Level Debugging of Combinational Circuits

In this section the MaxSat debug formulation for combinational circuits is presented. In order to express the debugging problem as a MaxSat problem, the circuit must first be converted into CNF. For the purpose of this paper we assume that circuits consist of single-output logic gates and CNF conversion occurs on a per gate basis in linear time [21], [22]. The goal of our debugging formulation is to identify a tuple of clauses in the CNF representation that are most likely responsible for the failure.

A. Partial MaxSat Formulation for Combinational Circuits

The Partial MaxSat formulation to debug a combinational circuit C given a correct specification is as follows:

$$\Phi = [I][O] \cdot CNF(C) \tag{2}$$

where $CNF(C)$ is the CNF representation of the erroneous circuit, I represents the input constraints, and O are the corresponding expected output constraints. The input and expected output constraints are modeled using hard clauses (as indicated by the square brackets) as their values are assumed to be correct. Since I excites the erroneous behavior of C which is eventually observed at the output, the actual output of C does not match the expected output O. The formula Φ is therefore inherently unsatisfiable.

A MaxSat solver finds an assignment to Φ such that the set of satisfied clauses is maximized. The complement of this set represents the minimum set of clauses whose removal makes Φ satisfiable. These clauses are the most likely error sources responsible for the unsatisfiability of the problem. For the remainder of this paper this complement set will be referred to as the MaxSat solution. Each of these clauses in turn can be mapped to a set of gates that could be the cause of the erroneous behavior. Note that these clauses can originate from the same or different gates.

Example 1 *Consider the circuit given in Figure 1. Figure 1(a) represents the correct circuit implementation according to the specification. In the erroneous circuit of Figure 1(b) the* NOR *gate at the output is mistakenly implemented using an* OR *gate. Given the input stimulus* $\{i = 1, j = 1, k = 0\}$ *the actual output of the erroneous circuit is* $c = 1$*. The expected output response is* $c = 0$*. The MaxSat debug formulation of this circuit expressed in CNF is given as follows:*

$$\begin{gathered}[i][j][\overline{k}][\overline{c}] \\ A\text{: } (i+\overline{a})(j+\overline{a})(\overline{i}+\overline{j}+a) \\ B\text{: } (\overline{k}+\overline{b})(k+b) \\ C\text{: } (\overline{a}+c)(\overline{b}+c)(a+b+\overline{c})\end{gathered}$$

For clarity we included the gates represented by each set of clauses in the above CNF.

Since the erroneous circuit cannot produce the expected response the problem is unsatisfiable. The maximum number of clauses that can be satisfied for this problem is 10 out of 12. A Partial MaxSat solution for this problem consists of two UNSAT clauses. One of the solution sets that may be returned by a Partial MaxSat solver is:

$$S_1 = \{C : (\overline{a}+c), C : (\overline{b}+c)\}$$

For clarity the gate to which each of the solutions map to is given before each clause. Both clauses in S_1 *correctly imply that somehow correcting gate* C *will fix the observed failure.*

We refer to this method of debugging as clause-level debugging. The number of clauses in the solution is known as the error clause cardinality of the solution. Clause-level debugging differs from SAT-based gate-level debugging techniques in that it seeks to find a set of erroneous clauses instead of a set of erroneous gates. However, the motivation behind both of these techniques is identical. Just as gate-level debugging finds erroneous gates in order to map them to bugs in the RTL code, clause-level errors can also be mapped to gates in the netlist or code in the RTL.

Clause-level debugging solves the debugging problem at a lower level of granularity since a single gate requires multiple clauses to specify its behavior. Intuitively this means that instead of the gate, rows in the gate's truth table are identified as erroneous. A single erroneous gate may result in multiple UNSAT clauses in the MaxSat solution. Consequently, while the cardinality of the clause-level solution is indeed minimal, the number of gates in the corresponding gate-level solution (its *gate-level cardinality*) might not be minimal.

For instance, in Example 1, the solver could have alternatively returned one of three other possible solutions:

$$\begin{gathered}S_2 = \{A : (\overline{i}+\overline{j}+a), C : (\overline{b}+c)\} \\ S_3 = \{C : (\overline{a}+c), B : (k+b)\} \\ S_4 = \{A : (\overline{i}+\overline{j}+a), B : (k+b)\}\end{gathered}$$

All solutions are of minimum error clause cardinality 2. However, the corresponding cardinality of gate-level solutions may vary. For S_2 (gates A and C), S_3 (gates B and C) and S_4 (gates A and B) the gate-level cardinality is 2 whereas for S_1 (only gate C) the gate-level cardinality is 1. The maximum number of UNSAT clauses due to a single gate is given by

the following theorem.

Theorem 1: Let G be a single-output logic gate that can drive both a value of 1 and 0 at its output. Then the maximum number of clauses that can be UNSAT in $CNF(G)$ is $m_g - 1$, where m_g is the number of clauses in the CNF representation of the gate.

Proof: Let y be the output variable of gate G. In order to force y to a value of 1, $CNF(G)$ must include a clause with literal y. Similarly, to force an output value of 0, one of the clauses in $CNF(G)$ must include the literal $\overline{y}$. Both the literals y and $\overline{y}$ must appear in $CNF(G)$ at least once and they cannot appear in the same clause. Therefore, assigning any value to y causes at least one clause in $CNF(G)$ to be satisfied. Thus MaxSat can return a maximum of $(m_g - 1)$ clauses per gate. ■

For instance, in Example 1, the behavior of gate A is represented by three clauses in the CNF. Thus $m_g = 3$ for A. The maximum number of clauses given any variable assignment to the CNF for gate A is 2.

B. Finding all MaxSat solutions

In practice, Φ has multiple solutions of minimum cardinality but the cardinality of the desired solution may not minimal. Conventional SAT-based debuggers find all solutions of a given error cardinality N_g to ensure completeness. N_g is defined as the maximum number of distinct gates contained in a solution. Similarly, clause-level debuggers need to find all solutions up to a maximum cardinality N_c. As with N_g the error clause cardinality N_c is the maximum number of clauses in Φ that are responsible for the bug. Existing MaxSat solvers can iteratively provide these solutions but a mechanism to *block* previous solutions is needed to avoid duplicates.

For solutions of cardinality one (i.e. the MaxSat solution only contains one clause) converting the single clause to a hard clause effectively blocks the solution. For cardinality $m > 1$ however, converting each clause to a hard clause would prevent these clauses from occurring in other solutions.

To illustrate this fact consider again Example 1. Suppose that the solution S_2 is found in the first MaxSat iteration. Then converting the clauses $(\overline{i} + \overline{j} + a)$ and $(\overline{b} + c)$ to hard clauses would not only block S_2 but also the solutions S_1 and S_4 since each of these solutions contains one of those clauses. Relevant solutions might be inadvertently overlooked depending on the order in which solutions are found.

Instead, the Partial MaxSat problem must be reformulated such that at least one of the solution clauses evaluates to true. To prevent a specific set of clauses $\{Cl_1, Cl_2, \ldots, Cl_m\}$ from being returned as a MaxSat solution, the following hard clause must be added to the CNF problem:

$$Cl_b = [Cl_1^l + Cl_2^l + \cdots + Cl_m^l] \tag{3}$$

In the above equation, we let Cl_i^l denote the sets of literals in the respective Cl_i clause. Since the set of literals in Cl_b is the union of literals from all the solution clauses, at least one of the clauses in $\{Cl_1, Cl_2, \ldots, Cl_m\}$ must be satisfied. For instance in Example 1 the solution S_2 can be blocked by adding the clause $[\overline{i} + \overline{j} + a + \overline{b} + c]$ to Φ.

All solutions of cardinality $\leq N_c$ can be identified by continuously blocking solutions of minimum cardinality. Using N_c we can guarantee that all solutions of a certain gate-level cardinality N_g are found. The relationship between the solutions obtained when using either N_c or N_g for combinational circuits is given by the following theorem.

Theorem 2: Let m_{cl} be the largest value of m_g for any gate in the circuit and let E_g be the set of all MaxSat solutions of gate-level cardinality N_g. Let E_c be the set of all solutions of given a maximum error clause cardinality $N_c = N_g \cdot (m_{cl} - 1)$. Then $E_g \subseteq E_c$.

Proof: By contradiction. Suppose that $S \in E_g$ is a solution such that $S \notin E_c$. Then $|S| > N_g \cdot (m_{cl} - 1)$ where $|S|$ denotes the number of clauses in S. However, the maximum number of clauses that can be UNSAT for a single gate is $m_g - 1$ and $m_{cl} \geq m_g$. Since S_c is of gate-level cardinality N_g, $|S| \leq N_g \cdot (m_{cl} - 1)$. Therefore $S_g \in E_c$ contradicting our initial assumption. ■

For Example 1, the highest value of m_g is 3 so $m_{cl} = 3$. To find all gate-level solutions of cardinality 1, N_c must be 2. Note that the gate-level cardinality of the solutions can vary between N_c and N_g.

IV. Clause Level Debugging of Sequential Circuits

A. Partial MaxSat Formulation for Sequential Circuits

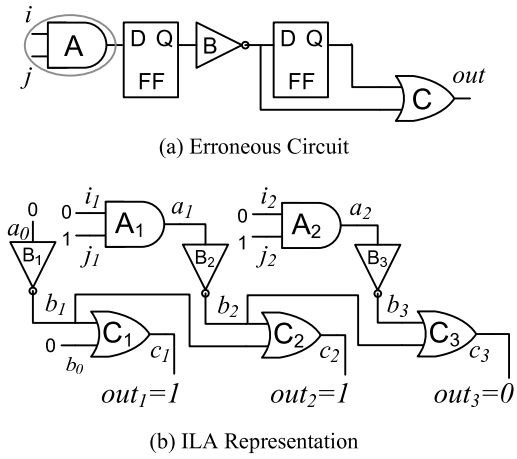
In this section the MaxSat formulation for combinational circuits is extended to sequential circuits. Our formulation takes as its inputs the sequential circuit C, a sequence of stimulus vectors $I_1, I_2, \ldots, I_k$ and expected output sequence $O_1, O_2, \ldots, O_k$. The variable k is the number of clock cycles in the error trace over which the behavior of C is modeled. The problem is also constrained by an initial state vector IS. The Iterative Logic Array (ILA) of the circuit, otherwise known as the time frame expansion model, is constructed by unrolling the combinational portion of the circuit k times. This effectively translates the sequential problem into a combinational one. The ILA can then be converted into a Boolean Satisfiability instance in CNF. The input to the MaxSat solver Φ is given by:

$$\Phi = \prod_{i=1}^{k} [I_i][O_i] \cdot [IS] \cdot CNF(ILA_k(C)) \tag{4}$$

where $ILA_k(C)$ denotes the time frame expansion of C for k time frames.

Example 2 *Consider the erroneous circuit in Figure 2(a). The correct circuit is derived by replacing gate A with an OR gate. The ILA representation of the circuit for three clock cycles is given in Figure 2(b). For clarity some irrelevant gates and constraints are omitted in the figure.*

The initial state of the circuit is given by $a_0 = 0$ *and* $b_0 = 0$. *The input vectors* $[i_1 = 0, j_1 = 1]$ *and* $[i_2 = 0, j_2 = 1]$ *cause the value at the fanout of gate* A *to differ from the correct values in time frames 1 and 2. The effects of these two error excitations are then propagated and observed in time frame 3 where the actual output (*$out_3 = 1$*) of the trace differs from the*



(a) Erroneous Circuit

(b) ILA Representation

Fig. 2. Erroneous circuit and its ILA representation

expected output ($out_3 = 0$). The MaxSat solutions for $N_c = 2$ are as follows:

$$S_1 = \{A_1 : (i_1 + \overline{a_1}), A_2 : (i_2 + \overline{a_2})\}$$
$$S_2 = \{A_1 : (i_1 + \overline{a_1}), B_3 : (a_2 + b_3)\}$$
$$S_3 = \{A_1 : (i_1 + \overline{a_1}), C_3 : (\overline{b_3} + c_3)\}$$
$$S_4 = \{B_2 : (a_1 + b_2), A_2 : (i_2 + \overline{a_2})\}$$
$$S_5 = \{B_2 : (a_1 + b_2), B_3 : (a_2 + b_3)\}$$
$$S_6 = \{B_2 : (a_1 + b_2), C_3 : (\overline{b_3} + c_3)\}$$
$$S_7 = \{C_3 : (\overline{b_2} + c_3), A_2 : (i_2 + \overline{a_2})\}$$
$$S_8 = \{C_3 : (\overline{b_2} + c_3), B_3 : (a_2 + b_3)\}$$
$$S_9 = \{C_3 : (\overline{b_2} + c_3), C_3 : (\overline{b_3} + c_3)\}$$

From these solutions, S_1 correctly implicates gate A as the source of the bug. The remaining solutions result from the presence of two propagation paths from the error excitations to the output signal out_3. One propagation path passes through the gates $A_1 \rightarrow B_2 \rightarrow C_3 \rightarrow out_3$ while the other path goes through $A_2 \rightarrow B_3 \rightarrow C_3 \rightarrow out_3$.

In the worst case the error could be excited in every clock cycle. Since a gate is replicated k times in $ILA_k(C)$ and the maximum number of clauses per gate is m_{cl}, an error clause cardinality of $N_c = N_g \cdot (m_{cl} - 1) \cdot k$ would be required to find all solutions of cardinality N_g. It is clear that the performance of the Partial MaxSat formulation depends on the number of error excitations where the erroneous behavior is propagated to an output. Previous work [23] show that errors in simulation traces are excited in temporal proximity to the outputs at which the bug is observed. From our experiments we also find that for the majority of cases the error is excited for only a few clock cycles (often only once or twice) before its effects can be first seen at the outputs. This experimental observation obviously works in favor of our formulation as MaxSat finds the least number of error excitations required to observe the bug.

Moreover, unlike conventional gate-level debuggers, our MaxSat formulation is not limited to finding a set of erroneous gates for a given error cardinality. Errors from the same or different gate-level sources are not distinguished at the clause level. For instance in Example 2 solution S_1 implicates two excitations of gate A as the cause of the problem. Solution S_2 indicates that an excitation of gate A followed by an excitation of gate B could be the bug. In both cases the number of excitations is two but no distinction is made whether these excitations come from the same or different gates. As a result, once an error clause cardinality N_c is specified, all clause-level errors are found irrespective of their corresponding gates. N_c can be more appropriately specified as:

$$N_c = N_{ep} \cdot (m_{cl} - 1) \tag{5}$$

where N_{ep} is the maximum *expected* number of gate-level error excitations and propagations for a given stimulus trace.

Note that N_{ep} and N_g describe two different concepts. N_g, describes an estimated number of spatial error locations that exist in the design irrespective of time frames or clock cycles. N_{ep}, however, describes an estimate of the maximum number of times the error is active in a particular error trace. In other words, the value of N_g relates to number of error location of the bug in the RTL, while N_{ep} relates to the observed erroneous behavior of a particular simulation trace. Similar to N_g, the user can provide an estimate for N_{ep} based on trace length and the complexity of the problem [8].

B. Extracting Temporal Information

An advantage of clause-level debugging is that both gate-level error sources and temporal bug information is provided. Since the circuit is replicated once for every clock cycle, each clause in the MaxSat solution of Φ represents an error location both spatially and temporally. Thus each solution clause automatically indicates when during the error trace the bug is active. For instance in Example 2, the clause $(i_1 + \overline{a_1})$ in S_1 indicates that gate A in time frame 1 could be one of the causes of the bug. The solution S_1 therefore states that gate A must be active in time frames 1 and 2 to cause the error.

Another advantage of the temporal information provided is that it establishes a more specific relationship between each of the gates in the solution. In gate-level debugging, when considering higher cardinality solutions, we can only determine that the locations in the solution act together to cause the error. However, the order in which each of these error locations act is unknown. Since temporal information is included in our MaxSat formulation, we can determine the sequence in which the individual gates are related. For instance solution S_2 in Example 2 indicates that an excitation of gate A in time frame 1 *followed* by an excitation of gate B in time frame 3 may be the cause of the error in the design. In contrast, traditional SAT-based debugging would only indicate that the gates A and B can act together to cause the bug.

Another approach is to aggregate the error information from all MaxSat solutions by creating a histogram showing the number of suspected error sources found for each time frame. The frequency of solutions per time frame can provide hints about when the errors are excited in the trace. The error is more likely to be propagated through circuit elements close to the actual error excitation so generally the time frames with the highest frequency of solutions are the best candidates. This aggregate temporal information can help the designer analyze the error trace [14]. Example graphs for different circuits are provided in Section VII for the Experiments.

V. Reducing MaxSat Iterations for all Solution MaxSat

A. Finding Additional Unsat Clauses

Since our formulation improves the granularity of the error model and finds time frame information, the search space for solutions is significantly larger compared to SAT-based debugging. Every gate is replicated over all time frames and each gate requires multiple clauses to represent. The number of MaxSat iterations required to find all solutions of cardinality $\leq N_c$ can be considerable. The purpose of this section is to improve on the technique given in Section III-B by applying a local search algorithm to explore the solution space [24]. In detail, we introduce a set of heuristics to derive more solutions based on existing MaxSat solutions that originate from previous iterations of the basic algorithm.

Consider again Example 2. The solutions S_1 to S_9 are due to the two propagation paths from the erroneous gate to the output. The respective variable assignments of each of these solutions only differ by a few variables. For instance, solution S_1 only differs from S_2 by the assignment to variable a_2. However, in order to find all of these solutions a total of nine MaxSat iterations is required. For larger circuits, each of these iterations can be computationally intensive.

Consider an UNSAT clause $cl_u \in \Phi$ for a given MaxSat assignment. Since cl_u is UNSAT each of its literals evaluate to 0. Inverting the variable assignment for one of its literals would cause cl_u to become satisfied. However another set of clauses $cls_i \subset \Phi$ with $|cls_i| \geq 1$ would become UNSAT. If this set only contains a single clause $cls_i = \{cl_s\}$ then the new assignment is another MaxSat assignment for Φ. The number of UNSAT clauses in the new assignment remains the same as in the original assignment and is therefore minimal. That is cl_u can be replaced with cl_s in the MaxSat solution since the solution cardinality remains unchanged. We say that cl_s is a substitute solution clause for cl_u.

An informal description of the technique applied multiple times to Example 2 is given below. Suppose the first MaxSat solution obtained is S_1. The clauses $(i_1 + \overline{a_1})$ and $(i_2 + \overline{a_2})$ are UNSAT. The variables a_1 and a_2 are therefore both set to 1 (i_1 and i_2 are constrained by hard clauses).

The clause $(i_1 + \overline{a_1})$ can be satisfied by setting $a_1 = 0$. As a result exactly one other clause, $(a_1 + b_2)$, becomes UNSAT. The clause $(a_1 + b_2)$ is therefore a substitute solution clause for $(i_1 + \overline{a_1})$ in S_1. Using this reasoning, the solution S_4 can be directly derived from S_1. Continuing from this new MaxSat assignment it is then possible to find another substitute solution clause, $(\overline{b_2} + c_3)$, by setting $b_2 = 1$. Similarly the clauses $(a_2 + b_3)$ and $(\overline{b_3} + c_3)$ can be found as substitute solution clauses for $(i_2 + \overline{a_2})$. These two sets of substitute clauses effectively identify S_2, S_3, S_4 and S_7 as additional solutions.

The algorithm to perform this local search is described in Algorithm 1. It takes an UNSAT solution clause cl_u, the MaxSat variable assignment va and the CNF problem Φ as its inputs. The output of this algorithm is a set of substitute clause and assignment modification pairs. An assignment modification mod consists of a subset of variables from Φ. Applying a modification mod to a variable assignment means that all the variables in mod are inverted to obtain a new assignment. In this case, the assignment modifications can be applied to va to obtain new MaxSat assignments.

Algorithm 1: The find_substitutes algorithm

Data: The CNF problem Φ, an UNSAT clause cl_u, and variable assignment va
Result: A set of substitute clauses and modifications to va for each clause

```
1  find_substitutes(Φ, cl_u, va)
2  begin
3      stack.push([cl_u, ∅])
4      subs ← [cl_u, ∅]
5      while stack ≠ ∅ do
6          [cl_t, mod] ← stack.pop()
7          invert_assignment(va, mod)
8          cls_c = get_connected_clauses(Φ, cl_t)
9          foreach literal l ∈ cl_t do
10             n ← 0
11             foreach clause cl ∈ cls_c do
12                 if sat_literals(cl, va) = {l̄} then
13                     n ← n + 1
14                     cl_s ← cl
15                 end
16             end
17             mod_l ← mod ∪ l
18             if n = 1 and (subs ∩ [cl_s, mod_l]) = ∅) then
19                 subs ← subs ∪ [cl_s, mod_l]
20                 stack.push([cl_s, mod_l])
21             end
22         end
23         invert_assignment(va, mod)
24     end
25     remove_hard_clauses(subs)
26     return subs
27 end
```

A stack is used by the algorithm to keep track of pairs that have not yet been examined by the algorithm. Whenever a new UNSAT clause cl_t is removed from the stack, a call to $invert\_assignments$ on line 7 modifies va such that cl_t is UNSAT under va. These modifications are reverted on line 23.

The search for solution clauses connected to cl_t is performed within the loop on line 9. The $get\_connected\_clauses$ function finds all clauses in Φ which have variables in common with cl_t. For each literal $l \in cl_t$ and every connected clause $cl \in cls_c$, the function $sat\_literals$ finds all satisfied literals in cl given va. If $\overline{l}$ is the only satisfied literal in cl, then inverting the variable assignment of l would convert cl into an UNSAT clause. If only a single clause $cl_s \in cls_c$ becomes UNSAT, then cl_s is a substitute solution clause for cl_t.

The set of substituted clauses found by the algorithm is stored in $subs$. To avoid revisiting previously found solution clauses, line 18 checks cl_s against previously found solutions before adding the new solution to the stack. Finally, line 25 removes any hard clauses from the result.

Note that the $find\_substitutes$ function is limited to find-

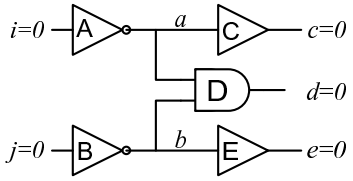


Fig. 3. Deriving solutions from substitute clauses example

ing MaxSat solutions which only differ by a single variable assignment from each other. Since we assume that our circuit consists of only single-output gates this method effectively finds clauses on an error propagation path consisting of functionally equivalent error locations.

B. Deriving More Solutions using Substitute Clause Sets

Using the function $find\_substitutes$ a new MaxSat solution can be derived for each new substitute clause found. In this section we examine how further solutions can be derived from combinations of substitute clauses.

Consider for example a cardinality two MaxSat solution: $\{cl_{sa}, cl_{sb}\}$ and suppose that cl_{ta} and cl_{tb} are substitute clauses for cl_{sa} and cl_{sb} respectively. Then based on the $find\_substitutes$ algorithm it is possible to obtain the solution $\{cl_{ta}, cl_{sb}\}$ by inverting a set of variables mod_a in the MaxSat assignment. Similarly, $\{cl_{sa}, cl_{tb}\}$ can be obtained by inverting a set of variables mod_b. From these observations we can deduce that inverting the set $mod_a \cup mod_b$ will convert $\{cl_{sa}, cl_{sb}\}$ into satisfied clauses and $\{cl_{ta}, cl_{tb}\}$ into UNSAT clauses. Thus $\{cl_{ta}, cl_{tb}\}$ is a MaxSat solution unless there exists some other clause that becomes UNSAT due to the inversion of $mod_a \cup mod_b$.

For instance the set of cardinality 2 solutions for Example 2 is essentially the cross product of the two substitute clause sets:

$$\{(i_1+\overline{a_1}), (a_1+b_2), (\overline{b_2}+c_3)\} \times \{(i_2+\overline{a_2}), (a_2+b_3), (\overline{b_3}+c_3)\}$$

The MaxSat assignment for each solution is obtained by combining their respective assignment modifications. Note that solutions obtained in this manner are not guaranteed to be valid MaxSat solutions. Consider the following example.

Example 3 *The CNF of the circuit given in Figure 3 is expressed as follows:*

$$\begin{gathered}[\overline{i}][\overline{j}][\overline{c}][\overline{d}][\overline{e}] \\ A\text{: } (\overline{i}+\overline{a})(i+a) \\ B\text{: } (\overline{j}+\overline{b})(j+b) \\ C\text{: } (a+\overline{c})(\overline{a}+c) \\ D\text{: } (a+\overline{d})(b+\overline{d})(\overline{a}+\overline{b}+d) \\ E\text{: } (b+\overline{e})(\overline{b}+e)\end{gathered}$$

Suppose that the initial solution is $S_1 = \{A : (i+a), B : (j+b)\}$. *Then finding the substitute clause* $(\overline{a}+c)$ *for* $(i+a)$ *by setting* $a = 1$ *will yield the solution* $S_2 = \{(\overline{a}+c), (j+b)\}$. *Similarly, setting* $b = 1$ *will yield* $S_3 = \{(i+a), (\overline{b}+e)\}$. *Setting both* $a = 1$ *and* $b = 1$*, however, increases the number of UNSAT clauses to three since* $(\overline{a}+\overline{b}+d)$ *is now UNSAT.*

Example 3 illustrates that combinations of substitute clauses may not be valid solutions. However, in practice most combinations of clauses derived in this manner are MaxSat solutions. To take advantage of this derivation method, we must check the validity of each derived solution. Since the number of clauses affected by the assignment modifications is generally small, the run time required should be negligible.

The optimized clause-level debugging algorithm to find all MaxSat solutions of cardinality N_c is given in Algorithm 2. After each iteration of the MaxSat solver, $find\_substitutes$ obtains a substitute clause set for each solution clause. The cross product of all substitute clause sets represents an over-approximation of alternative MaxSat solutions. The function $remove\_invalid\_solutions$ removes derived solutions whose assignment modifications increase the number of UNSAT clauses. On line 12 the derived solutions stored in E_t are then added to the solution set E_s. The function $block\_solutions$ blocks all solutions in E_t as described in Section III-B. The algorithm terminates once a MaxSat solution of cardinality higher than N_c is found.

VI. Debugging using Clause Groupings

Clause-level debugging can be a very powerful technique for locating temporal and spatial error locations that are excited infrequently. However, depending on the type of bug, the error cardinality required to find the relevant solution can be large. Increasing the error cardinality can reduce debugger performance since it is exponentially related to the complexity of the debugging problem [25]. If a large number of gates in the ILA representation are responsible for the error, a higher granularity error model might be required.

This section presents a technique for increasing the granularity of our error model by grouping a set of clauses such that they are treated as a single high level constraint. The concept of clause groupings is introduced for the general MaxSat case in [13]. Groupings can be created based on gates, modules and time frames. This allows us to increase the performance of our debugging algorithm at the cost of reducing the resolution of results. We can also use groupings to improve the performance of clause-level debugging by iteratively reducing the size of groups using hierarchical debugging from [15].

A. Creating Clause Groupings

From the MaxSat solver's point of view, the cost of setting all the clauses in the group to be UNSAT should be the same as the cost of allowing a single clause to be UNSAT. Groupings of clauses can be created by adding a single clause-selector literal to each clause in the group. To ensure that the problem remains UNSAT a unit clause consisting of the inverse of that literal is then added to Φ.

Consider for instance two clauses $cl_1 = (l_1 + l_2 + l_3)$ and $cl_2 = (l_4 + l_5)$ where $cl_1, cl_2 \in \Phi$. To create a grouping consisting of cl_1 and cl_2 the CNF is supplemented with an additional unit clause $(\overline{y})$. In the new CNF Φ' cl_1 and cl_2 are replaced with the following clauses:

$$\begin{aligned} cl'_1 &= (l_1 + l_2 + l_3 + y) \\ cl'_2 &= (l_4 + l_5 + y) \end{aligned}$$

Suppose Φ is satisfiable if cl_2 and cl_1 are removed. Then Φ' is satisfiable if the clause $(\overline{y})$ is removed since setting the $y = 1$ will allow both cl'_1 and cl'_2 to be satisfied.

Algorithm 2: The maxsat_clause_debug algorithm

Data: Partial MaxSat problem Φ and maximum cardinality N_c
Result: A set of suspect error sources E_s

```
1  maxsat_clause_debug(Φ, N_c)
2  begin
3      E_s ← ∅
4      [va, cls_s] ← solve_maxsat(Φ)
5      while |cls_s| ≤ N_c and cls_s ≠ ∅ do
6          E_t ← 1
7          foreach clause cl in cls_s do
8              subs = find_substitutes(Φ, cl, va)
9              E_t ← E_t × subs
10         end
11         remove_invalid_solutions(Φ, va, E_t)
12         E_s = E_s ∪ E_t
13         block_solutions(Φ, E_t)
14         [va, cls_s] ← solve_maxsat(Φ)
15     end
16     return E_s
17 end
```

Using this method to group connected gates, we can isolate regions in the circuit which could contain the bug. Bugs involving multiple clauses can therefore be identified without increasing the maximum cardinality. This technique effectively allows us to trade-off (*i.e.*, improve) run time at the expense of resolution. The granularity of the error model can be modified by varying the number of gates in each of these groups.

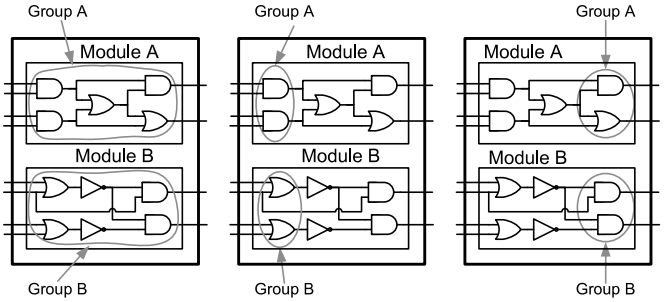
B. Groupings based on Gates

One application of groupings is to include clauses belonging to the same gate in a group. This effectively sets N_c to be the same as N_{ep}. Gate-level groupings for the circuit in Example 1 can be created as follows:

$$[i][j][\overline{k}][\overline{c}]$$
$$A\text{: } (i+\overline{a}+y_a)(j+\overline{a}+y_a)(\overline{i}+\overline{j}+a+y_a)$$
$$B\text{: } (\overline{k}+\overline{b}+y_b)(k+b+y_b)$$
$$C\text{: } (\overline{a}+c+y_c)(\overline{b}+c+y_c)(a+b+\overline{c}+y_c)$$
$$(\overline{y_a})(\overline{y_b})(\overline{y_c})$$

The variables y_a, y_b and y_c are the clause-selector variables. Removing any of the unit soft clauses will satisfy all the clauses in its respective gate.

One disadvantage of using gate-level groupings is that the $find\_substitutes$ algorithm from Section V cannot detect additional solutions if the number of clauses in the group that would otherwise be UNSAT is larger than 1. For instance for our above example, the first solution returned by the MaxSat solver is (y_c). Using the $find\_substitutes$ algorithm with $cl_u = (\overline{y_c})$ would yield no substitute clauses since setting $y_c = 0$ would unsatisfy more than one clause ($(\overline{a}+c)$ and $(\overline{b}+c)$). The benefit of gate groupings in this case is that they allow the Partial MaxSat solver to find the error location at a lower cardinality.



(a) Full Module Grouping (b) Module Input Grouping (c) Module Output Grouping

Fig. 4. Different implementations of module level groupings

C. Groupings based on Time Frame

For sequential circuits, groupings can be created not only according to circuit structure but also across time frames. Thus instead of creating a new group for each time frame, groups can span across fixed timing windows. The size of these timing windows represents a trade off between the accuracy of the temporal bug information and the complexity of the CNF problem. Ideally the window size should reflect the expected number of consecutive bug excitations to reduce the cardinality of the solution. If no temporal bug information is desired, a single window may encompass the entire trace.

D. Groupings based on Modules

A more sophisticated approach to groupings can be used for designs that are synthesized from RTL code. Hardware Description Languages (HDLs) such as Verilog and VHDL naturally group related design elements into modules or functions. Instead of identifying errors at the gate-level, module level groupings can be used to identify erroneous modules. These modules do not necessarily have to be explicit user defined modules but could also arise due to a simple add or subtract statement, or an if-else block. A combinational design consisting of two modules A and B is shown in Figure 4(a).

There are multiple approaches to creating a grouping for a module. The most intuitive way is to include all the gates of a module in a group as shown in Figure 4(a). We can derive an alternative encoding for module-level groupings if we consider that modules in an RTL design represent implications between the input and output signals of the module. Module level debugging operates under the premise that removing the input-output relationship imposed by the module would satisfy the problem. In order to remove this relationship from the circuit however, it is not necessary to remove every clause in the module. Removing only the clauses at either the inputs or the outputs of the module will achieve the same result. The remainder of the module then can be expressed as hard clauses to reduce the search space for the MaxSat solver.

Figure 4(b) and (c) illustrate groupings using input and output gates respectively. Empirically, we observe that grouping the gates at the module outputs yields the best performance results. While Figure 4 demonstrates module groupings for a combinational circuit, the technique can be easily extended to the sequential case. Time frame groupings can be used in conjunction with module groupings to improve efficiency.

Algorithm 3: The hierarchical_debug algorithm

Data: The circuit graph C, Partial MaxSat problem Φ, maximum clause-level cardinality N_c, maximum module-level cardinality N_m and maximum hierarchical depth L

Result: A set of module-level and clause-level error locations in the circuit

```
1  hierarchical_debug(C, Φ, N_c, N_m, L)
2  begin
3      module_solns ← ∅
4      clause_solns ← ∅
5      t ← top_level_module(C)
6      queue.enqueue(t)
7      while queue ≠ ∅ do
8          q ← queue.dequeue()
9          p ← nested_modules(C, q)
10         if level(q) ≤ L and p ≠ ∅ then
11             M ← module_debug(C, Φ, p, N_m)
12             module_soln ← module_soln ∪ M
13             foreach module m ∈ M do
14                 queue.enqueue(m)
15             end
16         else
17             c ← clause_debug(C, Φ, q, N_c)
18             clause_soln ← clause_soln ∪ c
19         end
20     end
21     return [module_solns, clause_solns]
22 end
```

E. Multiple Pass Debugging

Generally modules in digital designs are structured hierarchically with sub-modules nested inside parent modules. The simplest method to debug hierarchical designs is to create a new group for each nested module. To further boost performance, module-based hierarchical debugging [15], which iteratively traverses the module hierarchy by depth, can be used. Once a module-level solution of sufficient depth has been identified, clause-level debug can further refine the solution by identifying the exact erroneous gates and time frames.

The advantage of this hierarchical approach is two-fold. Firstly, since the intermediate results of this algorithm are suspect error modules, they can be immediately used by the designer for debug. Secondly, the search space for clause-level debugging is considerably smaller if only a few modules are being considered. Effectively, module-level debugging can act as a preprocess to speed up debugging for large circuits [10], [13]. Example 4 demonstrates how hierarchical debugging works for a circuit simulated for three clock cycles.

Example 4 *Consider the circuit given in Figure 5. For the first iteration (Figure 5(a)), two groups, one for each of the two top level modules, are created. The modules are grouped across all time frames, hence no temporal information is obtained from the solutions. Using Partial MaxSat, group G_A is then identified as the only solution at this level. Since we only wish to find solutions in module for A in the next iteration, module B is specified using only hard clauses.*

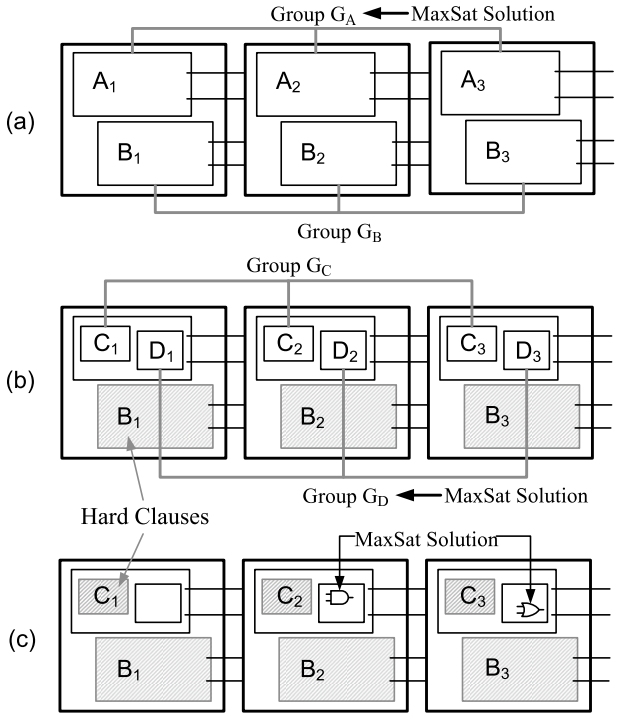


Fig. 5. Multiple pass debugging example

Suppose now that module A consists of the two sub-modules C and D. The groups G_C and G_D are created as shown in Figure 5(b). In this iteration module D is identified as erroneous. Consequently, the clauses in group G_C are also converted into hard clauses.

Note that the search space for each of these iterations consists of only two modules. If all modules were assigned a group on the first iteration and assuming module B contains no nested modules, the search space would consist of four groupings on the first iteration and three on the second.

Figure 5(c), shows the case where clause-level debugging is performed for the last iteration. Since only the clauses in module D are soft clauses, the solver will exclusively return solutions from that module. The last iteration also provides the temporal error locations through its solutions.

The hierarchical debugging algorithm which finds all module-level and gate-level solutions to an erroneous circuit C is given in Algorithm 3. The algorithm performs a traversal of erroneous modules in the design hierarchy using a queue starting with the top level module as shown on line 5. During each iteration of the loop, the algorithm finds the nested modules p to a set of erroneous modules q. The function $module\_debug$ on line 11 then performs module level debugging by creating output groupings for modules in p as described in Section VI-D. Similar to clause-level debugging, the MaxSat solver is then used to iteratively find all module level solutions $\subseteq p$ given the module-level cardinality N_m. A module level solution can be blocked by simply removing the grouping and converting all the clauses in the group to hard clauses.

TABLE I
PROBLEM SIZE AND RUN TIME COMPARISON BETWEEN SAT-BASED GATE-LEVEL DEBUGGING, CLAUSE-LEVEL DEBUGGING USING MAXSAT, AND GATE-LEVEL DEBUGGING USING MAXSAT

Error Trace			SAT-Based Debugging			Clause-Level Debugging						Gate-Level Groupings		
circuit	# gates	# time frames	# lits	# clauses	time in sec.	# lits	# clauses	% clause reduction	time in sec.	speed up	N_c	time in sec.	speed up	N_g
divider_1	6291	40	9454k	3734k	39.31	1830k	736k	80.29%	11.32	3.47	1	11.87	3.31	1
divider_2	6291	40	9641k	3807k	37.59	1869k	751k	80.27%	13.72	2.74	1	13.96	2.69	1
fpu_1	86020	40	153191k	60804k	2182.57	27424k	10313k	83.04%	462.12	4.72	3	250.78	8.7	1
fpu_2	87144	19	6013k	2441k	20.24	774k	316k	87.05%	8.64	2.34	2	6.43	3.15	1
hpdmc_1	18444	28	4677k	1859k	24.19	794k	319k	82.83%	3.39	7.14	1	4.64	5.21	1
hpdmc_2	18444	58	7464k	2896k	45.26	1724k	696k	75.97%	11.67	3.88	1	9.6	4.71	1
mem_ctrl_1	55174	40	92935k	37173k	358.45	14657k	5896k	84.14%	26.95	13.3	1	48.39	7.41	1
mem_ctrl_2	55174	40	96131k	38441k	422.53	15206k	6113k	84.10%	31.86	13.26	2	36.66	11.53	1
mips789_1	73600	32	131773k	53315k	606.4	17342k	6954k	86.96%	96.01	6.32	1	120.07	5.05	1
mips789_2	38524	158	8569k	3280k	47.04	2293k	957k	70.82%	278.51	0.17	1	49.36	0.95	1
mrisc_1	22452	42	40694k	16164k	361.79	6973k	2755k	82.95%	15.57	23.24	1	25.06	14.44	1
pipeline_1	5843	181	24776k	9306k	122.96	7039k	2779k	70.14%	22.52	5.46	1	63.71	1.93	1
pipeline_2	6318	69	14377k	5581k	70.25	3203k	1302k	76.67%	11.35	6.19	1	14.08	4.99	1
rsdecoder_1	15738	50	31771k	12623k	632.56	5458k	2250k	82.17%	17.87	35.4	1	26.76	23.64	1
rsdecoder_2	15732	100	49634k	19286k	509.02	10913k	4500k	76.67%	1616.14	0.31	1	912.37	0.56	1
spi_1	3427	14	3005k	1223k	19.76	351k	141k	88.47%	2.95	6.7	2	4.1	4.82	1
spi_2	3357	143	13697k	5207k	228.08	4036k	1635k	68.60%	142.45	1.6	1	489.97	0.47	1
sudoku_1	46668	61	102302k	40551k	435.44	18469k	7587k	81.29%	40.96	10.63	1	52.93	8.23	1
							Average:	**79.92%**	Average:	**4.49**		Average:	**4.04**	

Clause-level debugging (line 17) is performed instead of module-level debugging once the module depth exceeds a certain user defined value L or if the module does not contain any other sub-modules. The $clause\_debug$ function is a modified version of the $maxsat\_clause\_debug$ algorithm from Section V-B which also takes the set of modules q and the circuit graph C as its input. Only the gates in q are specified using soft clauses while the remainder of the circuit is expressed with hard clauses. Thus only solutions involving clauses from q are returned by the solver. Since the number of gates in the modules of q are a fraction of C's total number of gates, the run time per iteration is reduced.

VII. EXPERIMENTS

In this section we experimentally demonstrate the effectiveness of our debug techniques. The techniques described in this paper are implemented using C++ using the solver from [19] as the underlying MaxSat solver. All experiments are run on a 2.20GHz Intel Core2 Duo machine with 4GB of memory. For the remainder of this section, unless otherwise stated, averages are calculated by taking the geometric mean of the results.

In total two educational circuits (`pipeline`, `sudoku`) and eight circuits obtained from OpenCores.org [26] (`divider`, `fpu`, `hpdmc`, `mem_ctrl`, `mips789`, `mrisc`, `rsdecoder`, `spi`) are presented. A single Verilog bug is inserted into each circuit at the RTL level. These may include inverting the condition in an if-statement, changing the operator in an expression or modifying a state machine to transition to an erroneous state. In `rsdecoder_1` for instance the increment of a counter is changed from 1 to 2.

Each circuit is simulated using a testbench comparing the simulated output values with the expected output values of the circuit. As soon as an inconsistency is detected between the expected and observed values of the circuit, the simulation is terminated and the trace is recorded. The circuit is then synthesized and converted into CNF using the method in [21]. Some basic dangling logic removal is performed to reduce the problem size. The CNF is constrained using input and expected output values from the simulation of the correct circuit model as described in Section IV-A.

Table I compares the effectiveness of our clause-level (Section IV-A) and gate-level (Section VI) debugging technique against SAT-based debugging [8] with MiniSat2 [27]. The formulation of [8] most closely resembles our technique since results are returned in terms of gates. To allow for a closer comparison, gates are grouped across all time frames since results obtained in this manner are identical to the error locations obtained using [8] for a given N_g.

Columns 1 to 3 give the instance of the buggy circuit, the number of gates in the design, and the number of time frames in the ILA. Columns 4 to 6 then provide the number of literals, clauses, and the run time to obtain a single solution from of the debugging algorithm of [8].

The results for our MaxSat debugging algorithm are given in columns 7-12. The first two columns show the number of literals and clauses in the MaxSat formulation. The reduction in the number of clauses compared to [8] is given in the third column. The run time results for a single MaxSat iteration are given in the next column and the speed up compared to [8] is provided in column 11. The error clause cardinality is given in column 12. Finally, the results for our MaxSat formulation with gate-level groupings across all time frames are provided in columns 13-15.

Looking at instance `fpu_1` the number of gates is 86020 and the number of time frames in the error trace is 40. The problem formulation for gate-level SAT-based debugging consists of 153191 thousand literals and 60804 thousand gates. The time it takes to solve this problem with MiniSat is 2182.57 seconds. Expressing this debugging problem using our Partial MaxSat problem we see a reduction of 83.04% in the number of clauses. The time it takes to solve this problem with Partial MaxSat is 462.12 seconds, 4.72 times faster than the SAT-based solution. The error clause cardinality is 3. Using gate-level groupings, the run time is further reduced in this case to 250.78 seconds, an 8.7 times speed up. Note also that the

TABLE II
ITERATIVE SOLVING RESULTS

design	# iter	# soln	total time	soln/iter	time/iter
divider_1	10	65	111.22	6.5	11.12
fpu_1	3	68	1088.68	22.67	362.89
hpdmc_1	8	26	30.82	3.25	3.85
mem_ctrl_1	8	14	231.7	1.75	28.96
mips789_1	33	312	3049.83	9.45	92.42
pipeline_1	21	195	825.64	9.29	39.32
rsdecoder_1	2	2	49.68	1	24.84
			Average:	**8.02**	**32.78**

cardinality for this solution is reduced from 3 to 1 compared to clause-level.

Recall that for SAT-based debugging each gate is enhanced with a correction model (in the form of a multiplexer) and constraints are included in the CNF to limit the cardinality of the solution. The size of these constraints is quadratic with the number of gates in the circuit. Thus the number of clauses in our MaxSat formulation is considerably smaller in comparison. For the instances evaluated the size reduction is 80% on average.

It should be noted that the SAT-based debugging technique from [8] could be modified to also include time frame information. In the original formulation of [8] multiplexer select lines for the correction model are shared for the same gate across all time frames. As a result, the number of inputs to the constraints logic enforcing error cardinality is equal to the number of actual gates in the circuit. In order to obtain time frame information, however, each of these select lines needs to be a separate signal. This increases the number of inputs to the constraints circuitry to $(number\_of\_gates) * (number\_of\_time\_frames)$. We did not include any experiments for this modified SAT formulation because for most of our circuits the CNF problem could not be loaded into memory due to the increased size of the constraints logic.

Comparing against the original formulation [8], MaxSat consistently outperforms SAT-based debugging for short traces. For longer traces however, the run time improvements can show greater variability due to the increased difficulty of the problem. For instance, for `mips_789` and `rsdecoder_2`, whose trace lengths are 158 and 100 clock cycles respectively, MaxSat performs significantly worse than SAT. In the case of `spi_2`, whose trace length is 143 cycles, gate-level debugging using MaxSat is slower than SAT-based debugging even though clause-level debugging is still faster. On the other hand, `pipeline_1` still shows a significant improvement in run time a trace length despite a trace length of 181 clock cycles. The geometric mean of the speed up for all examples compared to SAT-based debugging [8] is $4.49\times$ and $4.04\times$ for clause-level and gate-level debugging respectively.

The number of clauses and literals for gate-level MaxSat is not given in the table but can be easily calculated. From Section VI, each clause in the group will have one additional literal. Each gate in the design will belong to its own group and one new unit clause is added for each gate since the group spans all time frames. Therefore, the number of literals is increased by the number of clauses and the number of gates in the design from clause-level formulation. The number of clauses is only increased by the number of gates in the design.

A. All Solution Partial MaxSat

Sections III-B and V discuss how all solutions for a given cardinality can be obtained. Table II summarizes the results for seven of our sample instances. Columns 2-4 give the number of MaxSat iterations, the number of solutions found and the total run time required to find all solutions. The average of the number of solutions per iteration and the run time per iteration are given in column 5 and 6 respectively.

Considering all the circuits in Table II, the number of solutions found per iteration is 8. The time required to run $find\_substitutes$ algorithm and verify the cross-product of substitute sets is negligible. Since each solution would have required a separate MaxSat iteration the total run time is significantly reduced. Our method is effective since errors are often propagated to a single fanout gate. In other cases, the value of the output can be changed by manipulating a single input without affecting the remainder of the circuit. For bugs with lower error clause cardinality and where the error propagates to multiple gates in its proximity, the method can be less effective.

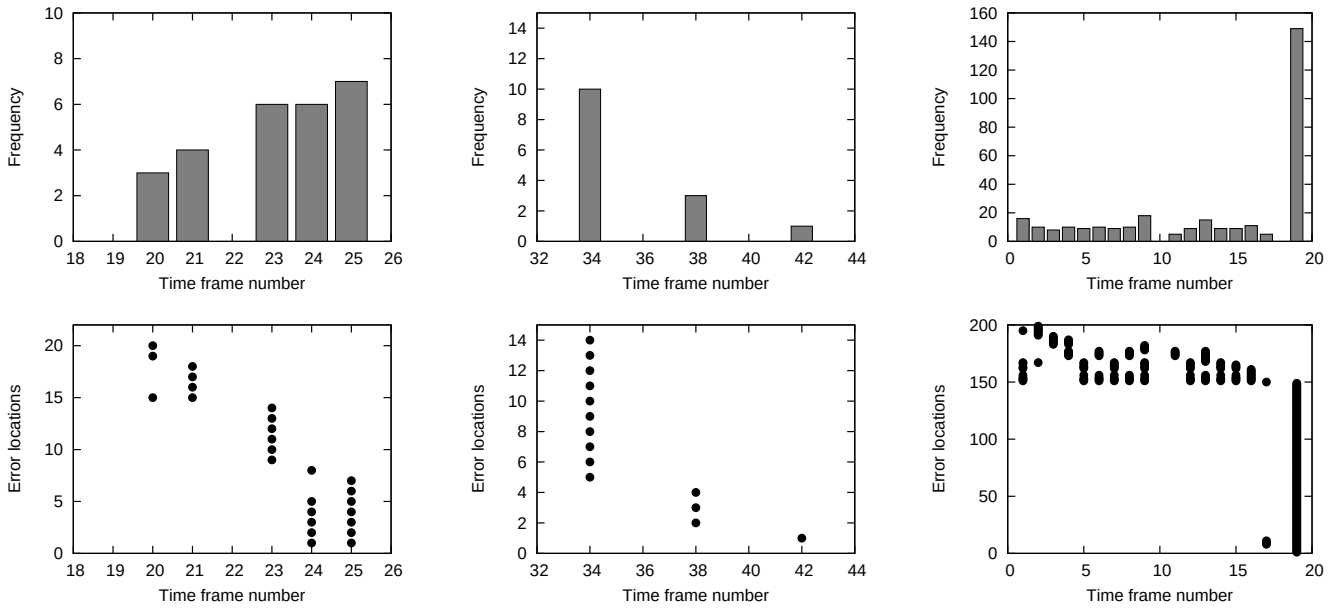
In our best case, all 68 solutions of `fpu_1` are found using only 3 MaxSat iterations. However, one reason why the number of solutions in this instance is large compared to other instances is related to the cardinality of the MaxSat solution. With an error clause cardinality of three, many of the solutions for `fpu_1` are different combinations of the same gates. The actual number of distinct error locations for `fpu_1` is 29.

B. Visualizing Temporal Information

As described in Section IV-B, providing temporal debug information is crucial in design debugging. The aggregate temporal information extracted by our technique for the circuits `mips789_1`, `hpdmc_1`, and `mem_ctrl_1` is illustrated in Fig. 6. The frequency each time frame is implicated by a solution clause is shown in the histograms in Fig. 6. The likelihood of an error being active during the time frame is indicated by the height of the bars. The scatter plots underneath the histograms plot error locations vs. the time frame for which they are found. The purpose of these is to illustrate which error locations are implicated multiple times for different time frames by the debugger. The y-axis lists all unique error locations found by the algorithm and the x-axis shows the time frames during which these locations can be excited to cause the error.

For instance the bug in `hpdmc_1` is created by removing a signal assignment in the RTL of a state machine. The actual error excitation occurs in time frame 24 since the removed signal was not assigned the proper value. The graph shows three highly likely erroneous time frames (23 to 25) and two less probable ones (20 to 21). For this case, a total of 20 distinct error locations are found. Many error locations in time frame 24 are also solutions in time frame 25.

For `mem_ctrl_1` one of the select signals of a MUX that controlled the data path is erroneously inverted. The solutions



(a) hpdmc_1 (b) mem_ctrl_1 (c) mips789_1

Fig. 6. Aggregate histograms and scatter plots for hpdmc_1, mem_ctrl and mips789_1

indicate that the bug could be fixed in time frame 34 when the error is excited before the data is propagated to the output. Since the data propagates through different gates in each time frame the solution gates are mostly unique as indicated in Figure 6(b). As the error propagates through the datapath the number of possible error sources also decreases. These graphs can allow the engineer to focus on specific regions in the design during specific time frames to correct the problem.

It is not always the case, however, that the height of the histogram provides the actual error excitation. The bug for `mips789_1` is created by changing the default assignment of a signal to a wrong value. From the scatter plot of Figure 6(c) we see that some error locations are implicated multiple times by the debugger until time frame 19. In this case examining these suspected locations instead of time frame 19 would be more beneficial. Nevertheless, the shape of the graph can provide valuable information to the engineer about the nature of the problem.

C. Grouping Clauses

Figure 7 depicts the average run time given different group sizes for all the circuits in Table I. Group sizes are given in terms of the number of gates in the group and groups span across all time frames. All the gates in the group are connected but groupings are created randomly. Figure 7 shows that the average run time generally decreases as the size of the group increases. The largest performance gain occurs when increasing the group size to 50 gates per group but the marginal gain is relatively small when increasing the size of groups further.

D. Hierarchical Debugging

Table III summarizes the results of our hierarchical debugging algorithm from Section VI-E. The table only shows run

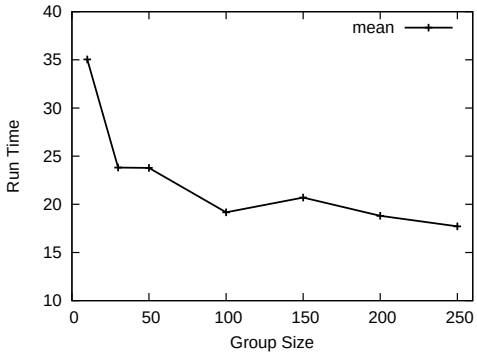


Fig. 7. Group Size vs Run Time

times for problem instances where the actual error location is located within a user defined sub-module of the design since the outputs of those are clearly defined. The purpose of these results is to compare the performance of clause-level debug with and without a hierarchical search. We show the time to find the actual erroneous module and time to find all clause-level error locations within that module. The number of gates in the targeted module is given in column 2. The time required to find the module using groupings as described in Section VI-D is given in column 3.

In our best example (`fpu_1`) the search space is reduced to 8320 gates (10% of total) and the search time is reduced by $5.4\times$. In all instances, the reduction in run time is sufficient to compensate for the additional time required to find the module. Comparing the average time per iteration from Table III against Table II a further speed up of $1.56\times$ is observed.

VIII. Related Works

Debugging techniques based on formal engines have garnered much attention from the research community since SAT-

TABLE III
HIERARCHICAL DEBUGGING RESULTS

design	Module Level Debug		Clause Level Debug of Module			
	# gates	search time	# iter	total time	time /iter	speed up
divider_1	5384	6.35	10	86.4	8.64	1.29
fpu_1	8320	2.89	2	133.92	66.96	5.42
hpdmc_1	6930	3.88	8	25.79	3.22	1.2
mem_ctrl_1	11035	25.24	7	198.59	28.37	1.02
mips789_1	66240	9.27	16	1141.12	71.32	1.3
pipeline_1	1578	12.33	17	446.32	26.25	1.5
rsdecoder_1	8499	60.4	2	36.82	18.41	1.35
				Average:	**21.04**	**1.56**

based debugging methodologies were introduced [28]. In order to extend the scalability of automated debugging techniques, extensions and alternative formal debugging methods have been proposed.

Many contributions to the debugging problem focus on reducing the size of the CNF problem and can also be applied to the MaxSat formulation presented here. In [10] for instance, unsatisfiable cores are used to speed up the debugging process for multiple fault diagnosis problems. This approach extracts a set of unsatisfiable cores from the CNF problem and prunes potential error locations not contained in any of the cores. A SAT-based exact debugger is then used to find the actual error locations from the reduced problem.

Another powerful technique to improve scalability is Abstraction Refinement [11]. The technique proposed in [11] "simplifies" design components according the design's structure and a pre-determined abstraction level. The debugging problem is then solved using the simplified model. Using the solutions returned, a refinement process selectively reintroduces abstracted components into the circuit and the process is repeated until no further refinements are necessary.

To deal with long error traces, trace compaction techniques [12], [29], [30] and interpolants [31] can be used to reduce the number of time frames in the ILA. Trace compaction techniques take as their input an error trace and attempt to generate an alternative sequence of stimulus events that expose the bug within fewer time frames. These shortened traces can then be used by a automated debugger to locate the actual error. In [31], the authors use interpolants to reduce the number of time frames in the ILA by replacing sequences of time frames with an over-approximation of their constraints. This allows for a partitioning of the problem into smaller sub-problems such that they can be solved more easily. The majority of these techniques can be applied to improve the scalability of the MaxSat formulation presented in this paper.

Other approaches rely on different solvers such as QBF [9], [15] and SMT [32], [33] to improve scalability. The module based debugging approach presented in Section VI-E is largely based on the work of [15], which combines QBF and hierarchical debugging. The QBF based debugging approach [9] offers the advantage of a much smaller problem formulation since only a single time frame is instantiated.

More recently, debugging formulations using SMT solvers have been proposed [32]. These formulate the debugging problem at the word level and use SMT solvers instead of SAT solvers to locate errors in the design. SMT solvers allow for the debugging problem to be formulated at a higher level of abstraction than Boolean. As with SAT-based debugging, QBF and SMT formulations for debugging do not provide any temporal debug information. Since these techniques use inherently different formal engines, they present alternatives to the proposed technique. Note that all alternative techniques tend to compare directly against the original SAT-based technique to provide a common reference.

IX. CONCLUSIONS

This work introduces techniques for debugging circuits using Partial MaxSat. We introduce an iterative method which accurately identifies all spatial and temporal error locations in a sequential design given an error trace. Compared to SAT-based debugging our formulation improves the granularity of spatial error locations by identifying the entries in the truth table of each gate as possibly erroneous. This paper also provides additional techniques to speed up the process of finding all MaxSat solutions. We describe a fast search algorithm that quickly identifies multiple substitute clauses and alternative solutions from a given MaxSat solution. A technique to group related clauses is presented to further improve performance using hierarchical debugging. Hierarchical debugging can quickly reduce the search space and therefore the solve time for our Partial MaxSat solver. The effectiveness of this method is demonstrated experimentally. For clause-level debugging we observe a $4.5\times$ improvement in run time and a 80% reduction in problem size compared to a conventional SAT-based approach. Our search algorithm can find 8 additional MaxSat solutions per iteration with little computational overhead. Hierarchical debugging further improves the performance of clause-level debug by focusing on specific modules in the design and reducing the search time per iteration by $1.56\times$.

REFERENCES

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publisher, 2000.

[2] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Correct Hardware Design and Verification Methods (CHARME)*, 2005, pp. 254–268.

[3] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Int'l Conf. on CAD (ICCAD)*, 2006, pp. 836–843.

[4] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Int'l Conference on Computer-Aided Verification (CAV)*, 2008, pp. 5–10.

[5] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing - an alternative to fault simulation," in *Design Automation Conf. (DAC)*, 1983, pp. 214–220.

[6] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[7] S.-Y. Huang, "A fading algorithm for sequential fault diagnosis," in *Proceedings of the Defect and Fault Tolerance in VLSI Systems (DFT)*, 2004, pp. 139–147.

[8] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD (TCAD)*, vol. 24, no. 10, pp. 1606–1621, 2005.

[9] 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 (ICCAD)*, 2007, pp. 240–245.

[10] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors." in *Great Lakes Symp. on VLSI (GLSVLSI)*, V. Narayanan, Z. Yan, E. Macii, and S. Bhanja, Eds. ACM, 2008, pp. 77–82.

[11] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 1182–1187.

[12] K. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *Int'l Conf. on CAD (ICCAD)*, 2005, pp. 1045–1051.

[13] S. Safarpour, M. H. Liffiton, H. Mangassarian, A. Veneris, and K. A. Sakallah, "Improved design debugging using maximum satisfiability," in *Int'l Conf. on Formal Methods in CAD (FMCAD)*, 2007, pp. 13–19.

[14] Y. Chen, S. Safarpour, A. Veneris, and J. Marques-Silva, "Spatial and temporal design debug using partial MaxSAT," in *Great Lakes Symp. on VLSI (GLSVLSI)*. ACM, 2009, pp. 345–350.

[15] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD (ICCAD)*, 2005, pp. 871–876.

[16] Max-SAT 2009, "http://www.maxsat.udl.cat," 2009.

[17] 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.

[18] Z. Fu and S. Malik, "On solving the partial MAX-SAT problem," in *Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, 2006, pp. 252–265.

[19] J. Marques-Silva and V. M. Manquinho, "Towards more effective unsatisfiability-based maximum satisfiability algorithms," in *Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, May 2008, pp. 225–230.

[20] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Test in Europe (DATE)*. ACM, March 2008.

[21] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD (TCAD)*, vol. 11, pp. 4–15, 1992.

[22] P. Manolios and D. Vroon, "Efficient circuit to CNF conversion," in *Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, 2007, pp. 4–9.

[23] S. Safarpour, A. Veneris, and F. Najm, "Managing verification error traces with bounded model debugging," in *ASP Design Automation Conf. (ASPDAC)*, 2010.

[24] L. Kroc, A. Sabharwal, C. P. Gomes, and B. Selman, "Integrating systematic and local search paradigms: A new strategy for MaxSAT," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 544–551.

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

[26] OpenCores.org, "http://www.opencores.org," 2009.

[27] N. Een and N. Sorensson, "An Extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, 2003, pp. 333–336.

[28] A. Smith, A. Veneris, and A. Viglas, "Design diagnosis using Boolean satisfiability," in *ASP Design Automation Conf. (ASPDAC)*, 2004, pp. 218–223.

[29] S. Safarpour, A. Veneris, and H. Mangassarian, "Trace compaction using SAT-based reachability analysis," in *ASP Design Automation Conf. (ASPDAC)*, 2007, pp. 932–937.

[30] Y. Chen, S. Safarpour, and A. Veneris, "Optimal trace compaction with property preservation," in *Midwest Symposium on Circuits and Systems (MWSCAS)*, 2009.

[31] B. Keng and A. G. Veneris, "Scaling VLSI design debugging with interpolation," in *Int'l Conf. on Formal Methods in CAD (FMCAD)*, 2009, pp. 144–151.

[32] F. Z. Saeed Mirzaeian and K.-T. Cheng, "RTL error diagnosis using a word-level SAT-solver," in *Int'l Test Conf. (ITC)*, 2008.

[33] A. Sülflow, R. Wille, G. Fey, and R. Drechsler, "Evaluation of cardinality constraints on SMT-based debugging," in *International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE Computer Society, 2009.

PLACE PHOTO HERE

Yibin Chen received a B.A.Sc. degree in computer engineering from the University of Waterloo, Canada, and a M.A.Sc. degree in computer engineering from the University of Toronto, Canada. He is currently working as a design engineer at Vennsa Technologies, Toronto, Canada. His research interests include design debugging, formal verification techniques and formal engines, including SAT, MaxSat, and SMT solvers.

PLACE PHOTO HERE

Sean Safarpour received the B.A.Sc. degree in computer engineering from the University of British Columbia, Canada, the M.A.Sc. and Ph.D. degrees in computer engineering from the University of Toronto, Canada. He is currently chief technology officer at Vennsa Technologies, Toronto, Canada, where he is in charge of research and development. He is the author of dozens of conference and journal publications and one book on automated debugging. His research interests include design debugging, formal verification techniques and formal engines such as SAT, QBF and SMT solvers.

PLACE PHOTO HERE

Joao Marques-Silva received a Engineer's and MSc degrees from Instituto Superior Tecnico (IST), Technical University of Lisbon, Portugal, in 1988 and 1991, a PhD degree in Electrical Engineering and Computer Science from the University of Michigan, Ann Arbor, MI, USA, in 1995, and the Habilitation in Computer Science from the Technical University of Lisbon in 2004. Joao Marques-Silva is currently Stokes Professor of Computer Science and Informatics at University College Dublin (UCD), Ireland. Before joining UCD he held appointments at the University of Southampton, UK, and at the Technical University of Lisbon, Portugal. His research interests include algorithms for constraint solving and optimization, and applications in formal methods, artificial intelligence, operations research and bioinformatics. Dr. Marques-Silva serves as associate editor of Integration: The VLSI Journal. He is a senior member of IEEE and a member of the ACM. Dr. Marques-Silva received the 2009 CAV award for fundamental contributions to the development of high-performance Boolean satisfiability solvers.

PLACE PHOTO HERE

Andreas Veneris received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1998 he was a visiting faculty at the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto where today he is an Associate Professor. His research interests include CAD for debugging, verification, synthesis and test of digital circuits/systems, and combinatorics. He has received several teaching awards and a best paper award. He is the author of one book and he holds three patents. He is a member of ACM, IEEE, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.