Bounded Model Debugging

Brian Keng, *Student Member, IEEE,* Sean Safarpour, *Member, IEEE,* and
Andreas Veneris, *Senior Member, IEEE*

***Abstract*—Design debugging is a major bottleneck in modern VLSI design flows as both the design size and the length of the error trace contribute to its inherent complexity. With typical design blocks exceeding half a million synthesized logic gates and error traces in the thousands of clock cycles, the complexity of the debugging problem poses a great challenge to automated debugging techniques. This work aims to address this daunting challenge by introducing the Bounded Model Debugging methodology that iteratively analyzes bounded sequences of the error trace. Two techniques are introduced in this methodology to solve this growing problem. The first technique iteratively analyzes bounded subsequences of the error trace of increasing size until the error is found or the entire trace is analyzed. The second technique partitions the error trace into non-overlapping bounded sequences of clock cycles which are each separately analyzed. A discussion of these two techniques is presented and a unified methodology that leverages the strengths of both techniques is developed. Empirical results on real industrial designs show that for large designs and long error traces the proposed methodology can find the actual error in 79% of cases with the first technique and 100% of cases with the second technique. In cases where the methodology is not used only 21% of cases are able to find the actual error. These numbers confirm the benefits of the proposed methodology to allow conventional automated debuggers to handle much larger real-life circuits.**

***Index Terms*—Debugging, Verification, RTL, Interpolation, VLSI**

I. Introduction

The relentless consumer appetite for innovative electronic devices places great demands on the performance of modern Very Large Scale Integration (VLSI) designs. An inevitable result of this trend is an increase in both their size and complexity. This leads to significantly greater costs in both design verification and the subsequent debugging process. Since the complexity of both these tasks is projected to increase by nearly two orders of magnitude in the next few years [1], the research community is challenged to generate new Computer Automated Design (CAD) verification and debug methodologies to meet the time-to-market demands and the current growth rate of the semiconductor industry [2].

Property and equivalence checkers [3], [4], assertion-based verification and functional coverage tools [5], and recent advances in powerful solving engines such as Binary Decision Diagrams (BDD), Boolean satisfiability (SAT), and Satisfiability Modulo Theories (SMT) [6]–[8] have allowed verification Computer Aided Design (CAD) tools [9]–[12] to achieve great strides into their ability to aid engineers in detecting functional design errors. Despite these developments, there has been relatively less work directed towards debugging the error once it has been detected. In this work, the term debugging represents the process that follows functional verification where a failure is detected and the engineer tries to pin-point the location of the error and fix it. This issue is increasingly becoming a bottleneck as it has been reported that debugging, a predominantly manual process today, consumes up to 60% of the total verification time [13]. Therefore, it is important to develop automated debugging tools to alleviate this manual resource-intensive process.

A central reason behind the complexity of debugging is the increasing design size and the length of counter-examples. Today, typical design blocks can exceed half a million synthesized logic gates. Further, counter-examples (*i.e.*, error traces) returned by dynamic or formal verification methodologies can be thousands of cycles long [14]. Most existing automated debugging solutions replicate the combinational part of the circuit for as many cycles as the counter-example requires before they operate [15]–[18]. Due to this, the debugging complexity of a sequential circuit directly depends on the number of cycles in the error trace [19] and it grows exponentially with the number of injected errors. Admittedly, to cope with this daunting challenge, engineers must invent new CAD techniques to alleviate the manual debugging pain.

This work aims to address these concerns by introducing a set of techniques, which are collectively referred to as *Bounded Model Debugging (BMD)*, to cope with the growing complexity of the design debugging problem. The central theme behind BMD is the use of bounded subsequences of the error trace called windows to manage the excessive error trace length of modern designs. Each window is iteratively analyzed starting from the cycle where the failure first occurs. By iteratively analyzing bounded windows instead of the entire error trace, a dramatic reduction in the memory and run-time requirements of the debugger are achieved.

The first BMD technique called *window expansion* [20] is motivated by the empirical observation that functional errors are often excited in close temporal proximity to their failure point. It stems from the practical experience of verification engineers who begin their analysis for clock cycles just prior to the failure in the error trace. Later, they progressively expand their search radius to larger windows of clock cycles

Manuscript received December 22, 2009; revised April 07, 2010. Current version published June 7, 2010. This paper was recommended by Associate Editor A. Kuehlmann.

Brian Keng is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (email: briank@eecg.toronto.edu).

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

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).

Copyright (c) 2010 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

near the failure until the error is found. Similarly, window expansion begins by constructing a small debugging instance for a bounded suffix window of an error trace containing the failure. Based on the solutions returned, the methodology determines whether to expand the bound of the suffix or if the results are complete. This process repeats until either the resources are exhausted or the algorithm terminates with complete results.

The second BMD technique called *window partitioning* [21] generalizes the window expansion approach for SAT-based debugging as it divides the error trace into non-overlapping bounded windows and analyzes each one separately. It begins this process by constructing a debugging instance for a bounded suffix window of the error trace. The solutions are analyzed to determine whether the results are complete, and if they are not, an over-approximation of that instance is generated in the form of an interpolant. Using the interpolant, the next bounded window in the sequence is analyzed by explicitly modelling that window and combining it with the interpolant from the previous iteration. By iteratively analyzing a fixed sized bounded window, it avoids the costly computation of explicitly modelling the entire error trace. This results in a significant reduction in the amount of resources needed which may come at a small sacrifice to the final resolution, that is, the number of suspect locations returned.

Both BMD techniques have their own strengths and trade-offs where one may be preferred over the other under certain conditions. A discussion of these techniques is presented that describes the conditions under which each technique should be used in an automated debugging flow. From these discussions, a unified methodology is developed that leverages their strengths and balances their inherent trade-offs.

An extensive set of experiments on large hardware designs with long error traces from both `OpenCores` [22] and industrial partners illustrates the benefits of this work. The window expansion and window partitioning techniques are able to successfully find the actual error in 79% and 100% of the instances respectively while previous work is only able to find the actual error 21% of the cases.

The remaining sections are organized as follows. Section II introduces background information and notation necessary to understand the contributions in this work. Section III describes motivation for the intuition behind the BMD methodology. Window expansion is introduced in Section IV and window partitioning is presented in Section V. Section VI presents a discussion of these two techniques as well as a unified BMD methodology. Experiments are presented in Section VII and Section VIII concludes this work.

II. Preliminaries

A. Notation and Definitions

The letters x, y and s refer to the set of primary inputs, primary outputs and state (memory) elements, respectively. Furthermore, x^i, y^i and s^i denote Boolean vectors in the i^{th} clock-cycle, or *time-frame*, of the sequential operation of a circuit. Similarly, x^i_j, y^i_j and s^i_j refer to the j^{th} indexed bit in the i^{th} time-frame. Finally, X^i, Y^i and S^i denote a predicate for the i^{th} clock cycle on the primary inputs, primary outputs and state elements, respectively. The behavior of a sequential circuit C can be described formally by a transition relation, $T(s^i, s^{i+1}, x^i, y^i)$, which is `true` if and only if given the current-state s^i, applying primary inputs x^i to C will generate primary outputs y^i and the next-state s^{i+1}.

An *error trace* of length $k+1$ returned by a dynamic or formal verification tool generated from the erroneous design, is denoted by $\mathcal{V}_0^k$. It consists of an initial state predicate, a vector of primary input predicates and a vector of *correct* or *expected* primary output predicates for clock cycles 0 to k, which can be written as:

$$\mathcal{V}_0^k = \langle S^0, \langle X^0, \dots, X^k \rangle, \langle Y^0, \dots, Y^k \rangle \rangle \quad (1)$$

The error trace can be thought of as a cycle-accurate counter-example containing an initial state and a sequence of input values as well as a sequence of expected output values generated from a high-level reference model (*e.g.* C or Matlab). By simulating the design with the error trace, we demonstrate a mismatch between the erroneous circuit and its expected behavior. Note that each of the predicates can be written as unit clauses representing the value for each of the respective signals in each time-frame.

A *window* of an error trace from clock-cycles p to q, is defined as a consecutive subsequence of an error trace, $\mathcal{V}_p^q = \langle S^p, \langle X^p, \dots, X^q \rangle, \langle Y^p, \dots, Y^q \rangle \rangle$ where S^p is calculated by applying the initial state predicate and the first p primary input predicates to the transition relation, *i.e.* simulating the erroneous circuit for p cycles. A *prefix window* ($\mathcal{V}_0^{p-1}$) and *suffix window* ($\mathcal{V}_p^k$) denote windows either starting at the first clock cycle or ending at the last clock cycle respectively. In our presentation, we will occasionally omit the term window and just use the terms of suffix or prefix instead.

For this work, we assume that the error is first observed in the last clock cycle of the error trace. If this is not the case, a shorter error trace can be trivially generated by taking the shortest prefix that exhibits the erroneous behavior.

B. SAT-based Design Debugging

Design debugging finds all error location *suspects* that explain the erroneous behavior of a design exposed by a given error trace [23]. The *error cardinality* N of a debugging problem refers to the number of distinct suspects returned in a solution by the method used. A debugging method is said to be *complete* for a given error cardinality if it returns all solutions whose functions can be separately modified to fix the erroneous behavior. Most existing methodologies for sequential circuits replicate the state transition of the circuit for the length of the error trace [19], a fact that imposes performance constraints because the complexity of debugging grows exponentially to the number of errors [15]:

$$(combinational\ circuitry\ *\#\ trace\ cycles)^{\#\ errors} \quad (2)$$

Historically, fault diagnosis techniques such as simulation, BDDs and path-tracing [23] were first used to tackle debugging. Complementary techniques such as trace minimization [14] are also developed to reduce the debugging complexity by minimizing the error trace before debugging. More recently, SAT-based methodologies are shown to exhibit significant performance advantages when compared to traditional techniques [15]. Following the original work, extensions using Quantified Boolean Formula (QBF) [19], maximum satisfiability [16], UNSAT cores [24] and others [25], [26] extend the benefits of the original methodology. With no loss of generality, this paper presents BMD in terms of SAT-based debugging. As such, we present some essentials that remain relevant to the work presented here.

SAT-based design debugging [15] encodes the design debugging problem into a SAT instance for each given error trace. Each satisfying assignment to this instance corresponds to a suspect that can correct the erroneous behavior in the design. This instance is created in several steps.

First, the transition relation is enhanced by introducing a *correction model* for each potential error location (gate, module etc.). Each correction model has an associated suspect variable, e_i, which works as follows: if $e_i = 1$ then the i^{th} potential error location is disconnected from its fan-in and becomes a free variable. This can be achieved either through adding multiplexors [15], or directly in conjunctive normal form (CNF). This enhanced transition relation is denoted by $T_{en}(s^i, s^{i+1}, x^i, y^i, e)$.

Next, T_{en} is unrolled as a time-frame expanded model for the length of the error trace, such that the next-state of time-frame i is connected to the current-state of time-frame $i+1$, and the error trace predicates are applied to the initial state, input and output variables. An error cardinality constraint $\Phi_N(e)$ is also added to denote the search for N errors. Given an error trace $\mathcal{V}_p^k$, debugging is encoded in SAT as follows:

$$\begin{aligned} Debug_p^k = & S^p \wedge \Phi_N(e) \wedge \\ & \Big(\bigwedge_{i=p}^{k} X^i \wedge Y^i \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, e)\Big) \end{aligned} \tag{3}$$

where a satisfying assignment provides a solution in the form of a set of suspects. Returned solutions can be used as blocking clauses to allow the algorithm to iterate multiple times and return all possible suspect error locations [15]. The following example illustrates the process described above.

Example 1 *Figure 1 shows a two time-frame expanded circuit of an erroneous two gate design with one state element. The correction models for each gate are denoted by* $\otimes$. *The suspect variables for these correction models are* e_1 *and* e_2 *corresponding to gates* g_1 *and* g_2 *respectively. The actual error inserted in the design is that the gate* g_2 *is be a buffer instead of an inverter. The error trace:*

$$\mathcal{V}_0^1 = \langle \overline{s_0^0}, \langle x_1^0 \wedge x_2^0, x_1^1 \wedge x_2^1 \rangle, \langle y_1^1 \wedge y_2^1 \rangle \rangle$$

demonstrates an erroneous behavior of the circuit. For $N = 1$, *a satisfying assignment for* $\{e_1, e_2\}$ *is* $\overline{e_1} \wedge e_2$.

C. Unsatisfiable Cores and Interpolants

Given a SAT instance ϕ in CNF that is unsatisfiable, an UNSAT core denoted by U, is a subset of clauses of ϕ which is also unsatisfiable. Modern DPLL [27] solvers can generate a proof of unsatisfiability along with a corresponding resolution graph that shows that a SAT instance is unsatisfiable [28]. The resolution graph demonstrates how clauses in ϕ can be combined to generate the empty clause.

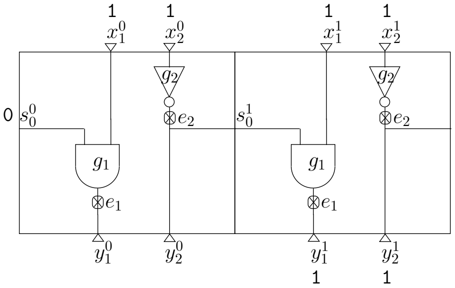


Fig. 1. Design unrolled for 2 clock cycles with correction models

An *interpolant* [29] is a Boolean formula that can be generated from a resolution graph. For a given unsatisfiable formula ϕ whose clauses can be partitioned into two subsets, A and B, an interpolant is a formula P, with the following properties:

(a) $A \to P$
(b) $B \wedge P$ is unsatisfiable.
(c) P only contains only common variables of A and B.

Formula P can be thought of as a particular over-approximation of the clauses in A that is UNSAT when combined with B and only involve variables common to A and B. Interpolants have been used in several applications including model checking [30] and synthesis [31]. There exists an algorithm [30] that can generate an interpolant as a Boolean circuit from the resolution graph. This algorithm can be computed in linear time in the number of vertices plus literals in the resolution graph.

III. Motivation

This section gives motivation for the BMD methodology by building intuition for the empirical observation that errors are usually located in close temporal proximity to their observation points. It begins with a probabilistic analysis of the likelihood that an error can propagate and be observed a given number of cycles later. By making assumptions about the probability of the error propagation and observability, the analysis is then simplified and presented graphically to gain intuition about this empirical observation.

It should be noted that this discussion uses a probabilistic argument which may not precisely model actual circuit behavior. It is also not used as a basis for the theory or mechanics of the BMD methodology described in later sections. Instead, the purpose of the analysis in this section is to reinforce the empirical observation that errors are usually located close to their failure points.

Proposition 1 *Assuming that a single error is excited in clock cycle* 1 *and no other errors are excited in any other clock cycles, let* $prop_i$ *be the probability of the error propagating from cycle* i *to* $i+1$ *and* obs_i *be the probability of observing* *a failure in clock cycle i, given that the error has propagated to that cycle. Also assume that the input vector sequences are temporally independent and stationary random sequences. Then, the probability of observing the first failure in clock cycle d is $p_d = \prod_{i=1}^{d-1} prop_i \times \prod_{i=1}^{d-1} (1 - obs_i) \times obs_d$.*

Proof: Let $W_i = \{$an error propagates from cycle i to cycle $i+1$ if it has propagated to cycle i $\}$, and $O_i = \{$a failure is observable in cycle i if an error has propagated to cycle i $\}$, and $E_1 = \{$an error is excited in clock cycle $1\}$. Probability p_d can be stated in terms of events W_i, O_i, and E_1: $p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \cap \bigcap_{i=1}^{d-1} \overline{O}_i \cap O_d \;\middle|\; E_1\right)$. By applying the identity $\mathcal{P}(A \cap B \mid C) = \mathcal{P}(A \mid C) \times \mathcal{P}(B \mid A \cap C)$, we get $p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) \times \mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} \overline{O}_i \cap \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$. Here, the events O_d and $\bigcap_{i=1}^{d-1} \overline{O}_i$ are conditionally independent of $E_1 \cap \bigcap_{i=1}^{d-1} W_i$. Thus, $\mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} \overline{O}_i \cap \bigcap_{i=1}^{d-1} W_i \cap E_1\right) = \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$. As a result, p_d can be simplified

$$p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) \times \mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right).$$

One of the assumptions made is that input vectors in successive cycles are all (temporally) independent. Thus, any W_i is independent of W_j for all cycles $i \neq j$: $\mathcal{P}(W_i \cap W_j \mid E_1) = \mathcal{P}(W_i \mid E_1) \times \mathcal{P}(W_j \mid E_1)$. As a result, $\mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) = \prod_{i=1}^{d-1} \mathcal{P}(W_i \mid E_1)$. Similarly, by the assumption, any O_i is independent of O_j for all cycles i and j:

$$\mathcal{P}\left(O_i \cap O_j \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right) = \mathcal{P}\left(O_i \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right) \times \mathcal{P}\left(O_j \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right).$$

As a result, $\mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) = \prod_{i=1}^{d-1} \mathcal{P}\left(\overline{O}_i \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right)$. Using the above, p_d can be simplified to:

$$p_d = \prod_{i=1}^{d-1} \mathcal{P}(W_i \mid E_1) \times \prod_{i=1}^{d-1} \mathcal{P}\left(\overline{O}_i \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right).$$

In the assumptions, $prop_j$ and obs_j are defined as: $prop_j = \mathcal{P}(W_j \mid E_1)$ and $obs_j = \mathcal{P}\left(O_j \;\middle|\; \bigcap_{i=1}^{j-1} W_i \cap E_1\right)$ for some cycle j. Using these definitions, p_d can be presented as

$$p_d = \prod_{i=1}^{d-1} prop_i \times \prod_{i=1}^{d-1} (1 - obs_i) \times obs_d$$

■

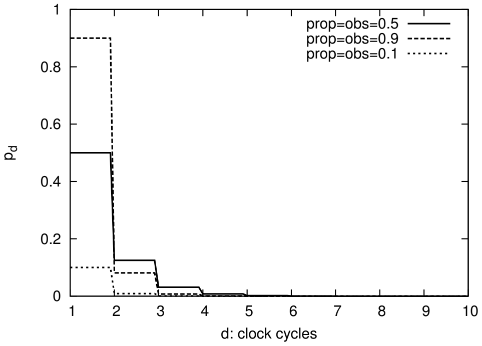


Fig. 2. Three curves of p_d as function of d

We can simplify p_d by assuming that $prop_i = prop$ and $obs_i = obs$ that remain constant for all cycles i resulting in $p_d = prop^{d-1} \times (1-obs)^{d-1} \times obs$. This simplified relationship is plotted in Figure 2 for three values of $prop = obs = \{0.1, 0.5, 0.9\}$. For values at $d = 1$ we have $p_d = P(O_1|E_1) = obs$. The negative exponential relationship is clear as the three curves are no longer visible when $d > 6$. Although overly simplified, the expression for p_d aligns with the observations made in the field and the experimental results of Section VII.

IV. Bounded Model Debugging with Window Expansion

Instead of debugging the entire error trace, one can apply any automated debugger to a suffix window of the error trace to generate a suffix debugging instance. Window expansion uses this fact to incrementally increase the size of this suffix window until the error is found. This key idea is illustrated in Figure 3. The figure shows an unrolled circuit for $k+1$ time-frames. The first iteration of window expansion starts from a suffix of the error trace and iteratively expands the window under analysis until the error is found. During each iteration, initial state suspect variables are used to determine if the current suffix can guarantee a complete result or whether a larger suffix must be analyzed.

The suffix debugging instance has the benefit of a reduced problem size due to the reduction in the number of time-frames analyzed, as also seen by Equation 2. In the worst case, window expansion can degenerate to a conventional debugging algorithm where the entire error trace has to be examined but experiments confirm that this is rarely the case.

It is clear that debugging a suffix instance can only find errors that are excited and propagated to the primary outputs within that particular suffix. However, suspects returned from

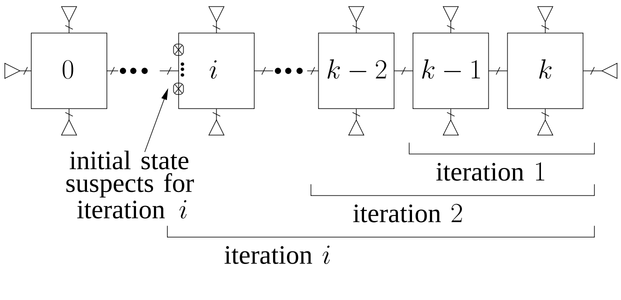


Fig. 3. BMD window expansion with multiple iterations

the suffix debugging instance have a useful property stated in the next lemma. This lemma guarantees that if a suspect is found during a suffix debugging instance, it is also a suspect to the entire error trace. This result is used in later sections to prove certain properties of the various BMD windowing techniques that we present.

Lemma 1 *For a given error cardinality N, any solution found when debugging a suffix of an error trace $\mathcal{V}_p^k$, will be found as a solution when debugging the entire error trace $\mathcal{V}_0^k$.*

Proof: Let E be a solution in the form of an assignment to the suspect variables in e such that for a suffix debugging instance, $Debug_p^k \wedge E$ is satisfiable. We wish to prove the lemma restated formally as follows:

$$Debug_p^k \wedge E \text{ is SAT} \rightarrow Debug_0^k \wedge E \text{ is SAT}$$

From Equation 3, we know that $Debug_0^{p-1} \wedge Debug_p^k$ and $Debug_0^k \wedge S^p$ generate the same clauses. $Debug_0^{p-1}$ is SAT regardless of the error trace because the failure has not been observed yet, so there is no mismatch in primary outputs. $Debug_0^{p-1} \wedge S^p$ is SAT when no suspect variables are active because the instance $Debug_0^{p-1}$ amounts to simulating the circuit for the first p cycles of the error trace generating the same values as S^p. Finally, $Debug_0^{p-1} \wedge S^p \wedge E$ is SAT because each active suspect variable allows the corresponding component outputs to become free, which will not change the satisfiability of an instance if it was already satisfiable.

Therefore, if $Debug_p^k \wedge E$ is SAT then $Debug_0^{p-1} \wedge Debug_p^k \wedge E$ is also SAT, since the only common variables are s^p and e which are fully assigned. As a result, $Debug_0^k \wedge S^p \wedge E$ is SAT implying that $Debug_0^k \wedge E$ is SAT as required. ■

A. Ensuring Completeness

Although Lemma 1 guarantees that a solution to a suffix is a valid solution for the entire error trace, it does not ensure that the set of solutions returned is complete. This is because the suffix may not contain the excitation of the error. However, if the error is excited in a cycle that precedes the current suffix, its effects must propagate through some state elements before reaching the observed failure at the primary outputs. We use this observation to ensure completeness for window expansion.

To determine whether or not the results are complete, we simply need to determine if changing the initial state predicate of the current suffix window will generate a satisfiable instance. This can be accomplished by introducing an *initial state suspect variable* for each state element. If an initial state suspect is returned by the debugger, then it indicates that the current suffix may not yield complete results and a longer suffix needs to be analyzed. For example in Figure 3 during the i^{th} iteration, if the debugger returns a solution including the initial state suspect (denoted by $\otimes$ in the figure) then the bounded window must be increased to ensure complete results are achieved. Alternatively, if the solutions returned contain no initial state suspects, we can safely conclude the results are complete.

The introduction of initial state suspects has an impact on the error cardinality which must be re-examined. The next theorem presents an upper bound on the error cardinality to guarantee the completeness of the approach.

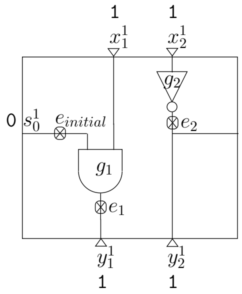


Fig. 4. Suffix window debugging with initial state suspects

Theorem 1 *Let $sols_N$ be the solutions returned by debugging the suffix window $\mathcal{V}_p^k$ for error cardinality N. Let $sols_{N+|s|}$ be the solutions returned by debugging the suffix window $\mathcal{V}_p^k$ with the addition of initial state suspects for error cardinality $N + |s|$, where $|s|$ denotes the number of state elements.*

If every solution in $sols_{N+|s|}$ does not contain any initial state suspects, then the solutions found when debugging the entire error trace $\mathcal{V}_0^k$ for error cardinality N will be exactly $sols_N$.

Proof: From Lemma 1, any debugging solution found for the suffix $\mathcal{V}_p^k$ will be found as a solution to the entire error trace $\mathcal{V}_0^k$.

We now prove by contradiction that if every solution in $sols_{N+|s|}$ does not contain any initial state suspects, then any solution found when debugging the entire error trace $\mathcal{V}_0^k$ will be found in $\mathcal{V}_p^k$.

Assume towards a contradiction that, no initial state suspects are found when the error cardinality is $N + |s|$ and that there is a satisfying assignment of suspect variables E that is found in $Debug_0^k$ but not in $Debug_p^k$. This means that E contains a positive assignment to some suspect variables which must have changed the excitation of the corresponding component in a cycle prior to the suffix $\mathcal{V}_p^k$. To affect an output in the k^{th} cycle, the component corresponding to the positive assignment of suspect variables must change the assignment to s^p that differs from the initial state predicate S^p for the suffix or else the instance would be unsatisfiable. However, if it changed the assignment to s^p, an initial state suspect would have been included in the solutions found when debugging the suffix $\mathcal{V}_p^k$ leading to a contradiction. So it must be the case that the solution would have be found in when debugging the suffix $\mathcal{V}_p^k$. ■

Example 2 *A suffix debugging instance derived from Example 1 is shown in Figure 4. The suffix, $\mathcal{V}_1^1$, is used to produce a suffix debugging instance $Debug_1^1$ with $N = 2$ and an initial state suspect $e_{initial}$. A satisfying solution consists of activating the suspect variables $\{e_{initial}, e_2\}$. According to Theorem 1, the results are not guaranteed to be complete and the suffix must be expanded.*

B. Performance Improvements

This section introduces two performance enhancements. The first relates to the error cardinality while the second one involves pruning the solution space. Since the difficulty of the debugging problem grows exponentially with the error cardinality as shown in Equation 2, it is important to reduce the size of this parameter. We can alleviate some of the complexity introduced by Theorem 1 by using a common initial state suspect variable for all of the initial state variables. In this way, we do not distinguish between different state variables. This reduces the error cardinality of the window expansion method as stated in the next corollary.

Corollary 1 *Let $sols_N$ be the solutions returned when debugging a suffix window $\mathcal{V}_p^k$ for error cardinality N. Let $sols_{N+1}$ be the solutions returned by debugging the suffix window $\mathcal{V}_p^k$ with the addition of a common initial state suspect for error cardinality $N+1$. If every solution in $sols_{N+1}$ does not contain the initial state suspect, then the solutions found when debugging the entire error trace $\mathcal{V}_0^k$ for error cardinality N will be exactly $sols_N$.*

Proof: This follows directly from Theorem 1, where instead of multiple initial state suspect variables, you have one that corresponds to the entire set of initial state variables. ■

The second performance enhancement uses Lemma 1 to reduce the problem size in later iterations by re-using results. During each iteration the debugger will find suspects for a given suffix. Lemma 1 states that every suspect found (except initial state suspects) is a suspect for the entire trace. This means that in future iterations we can safely ignore these suspects and reduce the search space.

C. Overall Algorithm

Algorithm 1 presents pseudo-code for the overall window expansion algorithm. The algorithm works by incrementally solving larger debugging instances by expanding the suffix window using the parameter $stride$, which can be chosen based on the design size to provide a trade-off between number of iterations and system resources. Initially, the algorithm uses the suffix window from clock cycle $k - stride$ to $k-1$. The BMD iterations are executed by the while loop on line 5 to line 14 where successive debugging problems are constructed with longer suffixes. On line 6 the suffix window debugging instance is created and solved. Once the solutions are removed and the initial state suspect is added as a potential error location, a new debugging instance is created with error cardinality set to $N+1$ on line 9. If this does not yield solutions that include the initial state suspect the algorithm exits with complete results as stated in Corollary 1. Otherwise, the algorithm removes the initial state suspect from consideration for the next iteration of the loop and expands the suffix window by $stride$.

Algorithm 1 BMD with Window Expansion

```
 1: $stride$ := expansion rate of algorithm
 2: procedure BMD_EXPANSION($stride$)
 3:     $e$ := set of potential suspect variables
 4:     $sols \leftarrow \emptyset$, $p \leftarrow (k - stride)$
 5:     while $p >= 0$ do
 6:         $sols_N \leftarrow sols_N \cup$ SOLVEALL$(Debug_p^{k-1}(N, e))$
 7:         $e \leftarrow e - sols_N$
 8:         $e \leftarrow e \cup e_{initial}$
 9:         $sols_{N+1} \leftarrow$ SOLVEALL$(Debug_p^{k-1}(N+1, e))$
10:         if $e_{initial} \notin sol$ for all $sol \in sols_{N+1}$ then
11:             return $sols_N$
12:         end if
13:         $e \leftarrow e - e_{initial}$, $p \leftarrow p - stride$
14:     end while
15:     return $sols_N$
16: end procedure
```

V. Bounded Model Debugging with Window Partitioning

BMD with window partitioning extends the results of the previous section to provide an attractive trade-off between performance and resolution for hard-to-debug problem instances. It does this by dividing the error trace into multiple non-overlapping bounded windows, iteratively analyzing each window separately. Each iteration directly models the current window and uses *interpolants* generated from previously analyzed windows to over-approximate the unmodelled suffix and examine a "sliding window" of clock cycles from the error trace.

The key idea is demonstrated in Figure 5 where it shows an unrolled circuit for $k+1$ time-frames. The error trace is partitioned into multiple fixed sized windows that are analyzed separately. In the first iteration, a standard suffix window of the error trace is used. If complete results are not guaranteed, the next non-overlapping window is then examined. However, in these subsequent iterations Equations 3 cannot be directly applied to the current partition because it does not model the erroneous behavior leading up to the observed failure. To solve this problem, an interpolant (P_{i+2}^k in Figure 5) is generated from previously solved iterations to over-approximate the unmodelled suffix to ensure the erroneous behavior is properly constrained. By only directly modelling a bounded window of the error trace in each iteration, a drastic reduction in runtime and memory is achieved at the cost of a potential loss in resolution. This window partitioning methodology is described in detail in the following subsections.

A. UNSAT Cores and Suffix Window Debugging

When debugging a suffix window, we can gain valuable information about suspects as stated by Lemma 1. However, there is also valuable information about the debugging problem in the resulting UNSAT core of a suffix debugging instance after all the solutions have been found. The intuition behind this idea is that if an UNSAT core exists in a suffix debugging instance that does not involve initial state variables, then it

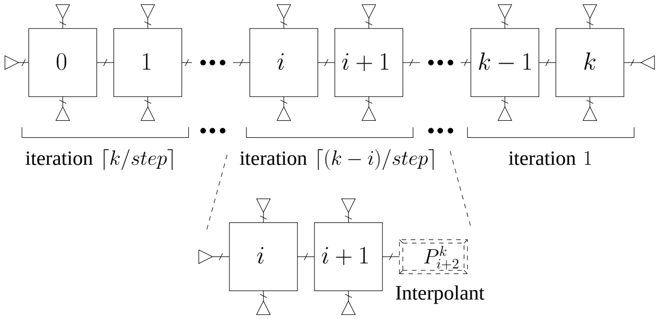

0
1
i
$i+1$
$k-1$
k
iteration $\lceil k/step \rceil$
iteration $\lceil (k-i)/step \rceil$
iteration 1
i
$i+1$
P^k_{i+2}
Interpolant

Fig. 5. BMD window partitioning

must exist in the entire debugging instance. This implies that the complete set of solutions have been found in the suffix debugging instance. The following theorem describes this property in more detail.

Theorem 2 *Let U be an UNSAT core generated after blocking all satisfying assignments to solutions for $Debug_p^k$. If $U \cap S^p = \emptyset$ then the suspects found in $Debug_p^k$ will be exactly the suspects found in the entire debugging instance $Debug_0^k$.*

Proof: From Lemma 1, any solution found in $Debug_p^k$ is a solution found in the entire debugging instance $Debug_0^k$.

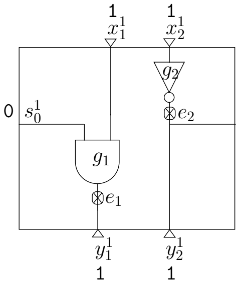
Now we prove by contradiction that any suspect found in $Debug_0^k$ will ALSO be found in $Debug_p^k$ if $U \cap S^p = \emptyset$. Assume towards a contradiction that E is an assignment to the suspect variables such that $Debug_0^k \wedge E$ is SAT and $Debug_p^k \wedge E$ is UNSAT. And let U be the UNSAT core derived after blocking all solutions to suspects for $Debug_p^k$, which contains no clauses in S^p.

Since $Debug_p^k \wedge E$ is UNSAT, E is not blocked by any of the blocked solutions (denoted by $blocked\_solutions_p^k$) to $Debug_p^k$. This means that $Debug_0^k \wedge E \wedge blocked\_solutions_p^k$ is satisfiable. However we know that in terms of clauses, $U \subseteq (Debug_p^k \wedge blocked\_solutions_p^k - S^p) \subseteq Debug_0^k \wedge blocked\_solutions_p^k$, since U does not contain any clauses from S^p. However, if $Debug_0^k \wedge blocked\_solutions_p^k \wedge E$ is satisfiable, then $U \wedge E$ is satisfiable. But U is an UNSAT core, which is a contradiction. So it must be the case that $Debug_p^k \wedge E$ is SAT. ■

Theorem 2 gives a condition to decide if debugging the current suffix window gives complete results. Notice that this argument does not depend on the error cardinality since both instances have the same cardinality. However in a similar manner to Theorem 1, if the UNSAT core contains state variables, to achieve complete results the prefix of the error trace must be analyzed. This is shown in the following example that illustrates a case where Theorem 2 cannot be applied.

Example 3 *A suffix debugging instance derived from Example 1 is shown in Figure 6. The suffix, $\mathcal{V}_1^1$, is used to produce a suffix debugging instance $Debug_1^1$ with $N = 1$. The clauses for the suffix debugging instance are shown to the right of the circuit diagram. This instance is unsatisfiable. The following is an UNSAT core from the instance:*

$$(x_2^1)(y_1^1)(y_2^1)(\overline{s_0^1})(\overline{e_1}+\overline{e_2})(\overline{x_2^1}+\overline{y_2^1}+e_2)(s_0^1+\overline{y_1^1}+e_1)$$



$$(\overline{s_0^1})(x_1^1)(x_2^1)(y_1^1)(y_2^1)$$
$$(s_0^1+\overline{y_1^1}+e_1)(x_1^1+\overline{y_1^1}+e_1)$$
$$(\overline{s_0^1}+\overline{x_1^1}+y_1^1+e_1)$$
$$(x_2^1+y_2^1+e_2)(\overline{x_2^1}+\overline{y_2^1}+e_2)$$
$$(e_1+e_2)(\overline{e_1}+\overline{e_2})$$

Fig. 6. UNSAT core from suffix window debugging

Using Theorem 2, we know that $Debug_1^1$ does not result in the complete set of suspects to $Debug_0^1$ because the UNSAT core contains the clause $\overline{s_0^1} \subseteq S^1$, so the prefix of the error trace still needs to be analyzed.

B. *Prefix Window Debugging*

Debugging a prefix window of an error trace requires additional consideration. Since the failure occurs in the last time-frame, modelling a prefix of the error trace according to Equation 3 will result in a trivially satisfiable instance since all the primary outputs will match the expected outputs in the error trace. To ensure that the observed erroneous behavior in the last time-frame constrains the prefix debugging instance properly, the prefix debugging instance is formulated in two parts. The first part uses the conventional SAT-based formulation (Equation 3) using a prefix of the error trace and the second part uses an interpolant approximating time-frames for the corresponding suffix of the error trace.

For a suffix debugging instance, an interpolant can be generated by partitioning the clauses into two subsets A and B as follows:

$$\begin{aligned} A =& \bigwedge_{i=p}^{k} X^i \wedge Y^i \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, e) \\ B =& S^p \wedge \Phi_N(e) \wedge blocked\_solutions \end{aligned} \tag{4}$$

Intuitively, A represents the enhanced transition relation as well as the primary input and output values for each time-frame of the suffix window. Subset B represents the initial state predicate, cardinality constraint and blocked solutions. The common variables are exactly the initial state and suspect variables. Using this partition an interpolant P_p^k can be generated from the resolution graph using the algorithm from [30].

The interpolant can be interpreted as an over-approximation of the suffix debugging instance that retains the cause of unsatisfiability *i.e.* the observed failure. It captures how the observed failure relates to the state and suspect variables. The benefit of using P_p^k is that, in most cases, the interpolant will be significantly smaller than explicitly modelling the time-frames. In cases where it is larger, the time-frames can be used directly, bounding the size of the problem. However, experimental results confirm that the interpolant is smaller than explicitly modelling the circuit.

Example 4 *Figure 7 shows the resulting resolution graph on the left and interpolant on the right from the UNSAT core in*

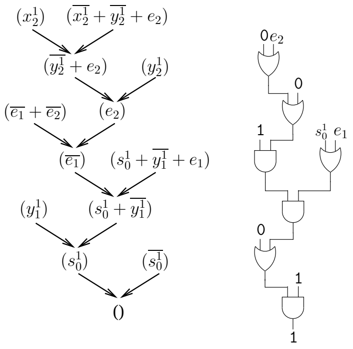

(x_2^1)
$(\overline{x_2^1} + \overline{y_2^1} + e_2)$
$(\overline{y_2^1} + e_2)$
(y_2^1)
$(\overline{e_1} + \overline{e_2})$
(e_2)
$(\overline{e_1})$
$(s_0^1 + \overline{y_1^1} + e_1)$
(y_1^1)
$(s_0^1 + \overline{y_1^1})$
(s_0^1)
$(\overline{s_0^1})$
$()$
0 e_2
0
1
s_0^1 e_1
0
1
1

Fig. 7. Resolution graph and interpolant

Example 3. Notice how many of the root nodes of the resolution graph generate constants values in the interpolant. This is a common occurrence and generally leads to a small interpolant relative to the UNSAT core that it was derived from.

By combining the interpolant with Equation 3, we can now construct a prefix debugging instance that only requires explicitly modelling the prefix window of the error trace. The interpolant will constrain the final states and suspect variables of the modelled prefix window implicitly constraining the failure. The prefix debugging instance constrained with the interpolant, $DebugItp_0^{p-1}$, can be written as follows:

$$DebugItp_0^{p-1} = Debug_0^{p-1} \wedge P_p^k \tag{5}$$

$DebugItp_0^{p-1}$ will be UNSAT when no suspect variables are active because $Debug_0^{p-1}$ will be equivalent to simulating the design for clock cycles 0 to $p-1$ and will implicitly generate the initial state predicate S^p which is known to be UNSAT with P_p^k.

The next example builds from previous ones to show how a prefix debugging instance can be created. However, note this example does not show the benefit of using interpolants to reduce the size of the problem due to its simplicity.

Example 5 *Figure 8 shows how the interpolant generated in Example 4 can be used to debug a prefix of an error trace. Notice that the interpolant is significantly smaller than the resolution graph once the constants propagate through the gates. In Figure 8, activating suspect variable e_2 leads to the only satisfying assignment. This is consistent with the solution found in Example 1.*

Since P_p^k is an over-approximation of the suffix, $DebugItp_0^{p-1}$ will not miss any solutions compared to debugging the entire error trace as stated in the next theorem.

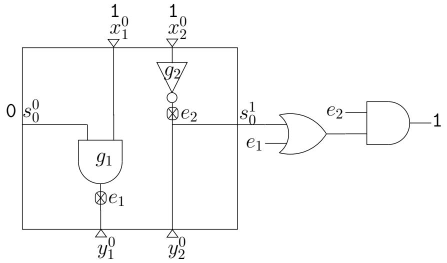


Fig. 8. Prefix window debugging with an interpolant

Theorem 3 *Any suspect found in $Debug_0^k$ will be found in $DebugItp_0^{p-1}$.*

Proof: By definition, $Debug_0^k = Debug_0^{p-1} \wedge A$, where A is defined in Equation 4. So any satisfying assignment to $Debug_0^k$ will satisfy $Debug_0^{p-1}$ and A. But $A \rightarrow P_p^k$, so it also satisfies P_p^k satisfying $DebugItp_0^{p-1}$. ■

Theorem 3 guarantees that solving the prefix debugging instance will not miss any suspects, however it may be the case that extra solutions will be returned that will not be found when debugging the entire error trace. This is due to P_p^k being an over-approximation of the suffix which may not constrain the prefix as strongly as explicitly modelling time-frames p to k directly. However, the relative increase in the number of extra solutions is usually small as seen in the experimental results.

C. Overall Algorithm

By using the ideas of suffix and prefix debugging, it is possible to further divide the debugging problem into smaller windows that model a user-defined number of time-frames. Algorithm 2 presents pseudo-code for a scalable debugging algorithm that divides an error trace of length k into $\lceil k/step \rceil$ windows, where $step$ is a user-defined parameter that specifies the maximum number of simultaneous time-frames to be directly modelled. This parameter can be chosen based on the design size to balance the number of iterations and system resources.

The algorithm iteratively analyzes windows of the error trace, starting with a suffix (lines 5-15). In each iteration, it begins by analyzing the current window of the error trace and finds all suspects shown on line 8. It then generates an UNSAT core and checks if it contains initial state variables. If it does not, it returns the current set of suspects (lines 9-12) and allows for an early exit with complete results (Theorem 2). If an early exit is not taken, it generates an interpolant from the UNSAT core. Finally, it prunes the search space by removing suspects found in this iteration.

Algorithm 2 provides a complete method to analyze long error traces that would not otherwise be possible with other techniques. However, there is a trade-off between the $step$ parameter and the final resolution. Each successive interpolant generated will potentially be a weaker constraint than the previous one. By setting $step$ to a small value, too many solutions can be returned. One way to cope with this is to provide a ranking of the solutions to the user so they can concentrate their effort on the most likely solutions. Algorithm 2 implicitly gives a useful ranking of solutions. More confidence can be given to solutions found in earlier iterations because a stronger constraint is used for the approximation of the suffix. In the case of the first iteration, all solutions found in the suffix will be found when debugging the entire error trace, as stated in Lemma 1.

Algorithm 2 BMD with Window Partitioning

```
 1: $step$ := maximum number of time-frames
 2: procedure BMD_PARTITION($step$)
 3:     $e$ := set of potential suspect variables
 4:     $sols \leftarrow \emptyset$, $P \leftarrow \mathtt{1}$
 5:     while $k > 0$ do
 6:         $p \leftarrow max(k - step, 0)$
 7:         $inst \leftarrow Debug_p^{k-1}(N, e) \wedge P$
 8:         $sols \leftarrow sols \cup$ SOLVEALL($inst$)
 9:         $U \leftarrow$ EXTRACTUNSATCORE($inst, sols$)
10:         if $U \cap S^p = \emptyset$ then
11:             return $sols$
12:         end if
13:         $P \leftarrow$ GENERATEINTERPOLANT($U$)
14:         $e \leftarrow e$ - $sols$, $k \leftarrow k$ - $step$
15:     end while
16:     return $sols$
17: end procedure
```

VI. A Unified BMD Methodology

This section provides a discussion on the benefits, trade-offs, and application of the window expansion and window partitioning techniques under a unified BMD methodology. It begins by analyzing each technique and describes the situations in which they are appropriate. This is followed by a discussion of a unified BMD methodology that leverages the strengths of both techniques.

The window expansion technique incrementally formulates the debugging problem by examining suffix windows of increasing size allowing for incremental user feedback. It is a complete method that provides significant benefit in terms of memory and run-time especially in earlier iterations when the suffix window is small. In later iterations where the suffix window is large, this benefit is reduced and in the worst case can degenerate to analyzing the full error trace which may require excessive resources.

The window partitioning technique on the other hand, divides the error trace up into non-overlapping partitions and separately examines each partition using interpolants to approximate the unmodelled suffix. It provides the ability to fully analyze long error traces because each partition is much smaller than the full error trace. However, due to the use of the interpolant as an over-approximation this technique may result in a loss of resolution causing too many suspects to be returned.

Both these techniques should be used in situations that leverage their strengths. Window expansion is best utilized as the default methodology for an automated design debugging flow because it provides complete results for each suffix it analyzes. Since many real-world errors are likely to be close to their failure points, this methodology will find many errors quickly with good resolution. In cases where window expansion exceeds the system resources, the only alternative is to use the window partitioning technique to analyze long error traces and find “deep” bugs even though there might be some potential loss in resolution. Using these two guidelines, the next section develops a unified BMD methodology that leverages the strengths of both techniques.

A. Overall Algorithm

By combining the window expansion and window partitioning techniques, a unified methodology can be developed that provides a more favourable trade-off compared to using the techniques separately.

The unified methodology begins by dividing the error trace into partitions using the window partitioning technique. For each partition, instead of using Equation 3 directly on the partition, it uses the window expansion technique to incrementally solve each one of them. Using both BMD techniques together provides a more flexible trade-off between resolution and resources. It addresses the two major challenges with using the BMD techniques in isolation. It addresses the first challenge by using the window partitioning technique to overcome the resource limitation of the window expansion technique. The second challenge is addressed by providing an incremental window expansion formulation to find an exact set of solutions for each partition.

The guidelines for the unified methodology are similar to the guidelines for the individual BMD techniques. The partition sizes should be as large as possible to ensure that the over-approximation does not return too many solutions. The window expansion technique is utilized on each partition to provide quick incremental results back to the user. In this situation, the number of times the interpolant is used as an approximation is minimized so that the resolution does not suffer, while the window expansion technique still provides the benefit of an incremental formulation for each large partition.

Algorithm 3 presents pseudo-code for the unified BMD algorithm combining both window expansion and window partitioning. The algorithm is passed two parameters, $step$ and $stride$, that control the expansion rate and the maximum number of time-frames for each partition. Each iteration of the loop on line 6 examines a window of time-frames corresponding to each partition. This process is similar to Algorithm 2. The difference lies on line 8 where BMD with window expansion runs on the current partition instead of directly solving Equation 3. By using the window expansion algorithm, each partition is solved incrementally gaining the benefit of both the window partitioning and window expansion algorithms.

VII. Experiments

In this section, we present experimental results for the proposed BMD methodology. All experiments are run on a single core of a Core 2 Quad 2.66 Ghz machine with 8GB of memory with a timeout of 3600 seconds. The debugger used is a C++ sequential SAT-based engine based on [15] with a Verilog frontend to allow for RTL-based diagnosis. MINISAT-V1.14 [32] is used to solve all the SAT instances as well as generate the proofs of unsatisfiability.

The effectiveness of the proposed techniques are shown on industrial Verilog RTL designs from OpenCores [22] as well as two commercial designs in production (`fxu`, `rx_comm`)

Algorithm 3 Unified BMD Algorithm

```
$step$ := maximum number of time-frames
$stride$ := expansion rate of algorithm
procedure UNIFIED_BMD($step$, $stride$)
    $e$ := set of potential suspect variables
    $sols \leftarrow \emptyset$, $P \leftarrow 1$
    while $k > 0$ do
        $p \leftarrow max(k - step, 0)$
        $sols \leftarrow sols \cup$ BMD_EXPANSION$_p^{k-1}(stride)$
        $P \leftarrow$ GENERATEINTERPOLANT$(U)$
        $e \leftarrow e$ - $sols$
    end while
    return $sols$
end procedure
```

TABLE I
DESIGN CHARACTERISTICS

Design Name	# gates	# DFFs	total # suspects
ac97	25314	2346	1092
div64bits	75111	5512	143
fdct	377849	5717	4658
fpu	81303	715	939
fxu	602713	29080	33088
mem_ctrl	46425	1145	2451
rsdecoder	11380	521	1623
rx_comm	586695	30339	15840
spi	2103	132	223
vga	153536	17055	1337
wb	3617	90	407

provided to the authors by semiconductor firms. Table I presents the design statistics. The four columns show the design name, number of total gates, number of flip flops and the total number of potential error locations.

Each debug instance is generated by randomly selecting an RTL line and inserting a typical industrial RTL error such as a wrong state transition, incorrect operator or incorrect module instantiation that is detected by the corresponding design testbench. The error trace is recorded from the simulation of the erroneous design with its testbench and suspects returned correspond to an error location in the RTL. It is important to emphasize that each RTL suspect may correspond to dozens of error locations in a gate-level formulation of the debugging problem. Effectively, multiple gate-level errors are implicitly detected using this setup.

A. BMD with Window Expansion

The experimental results for the proposed window expansion technique are presented in Table II. For each design in Table I, an instance is created by inserting an error into the design. Instances are named by appending a number after the design name to indicate which error is inserted. The first two columns of the table show the instance name and the number of clock cycles in the entire error trace. Each row of the table corresponds to an instance that is run using both a stand-alone debugger and the window expansion technique.

The next four columns of the table show the results of running the stand-alone SAT-based debugger using the entire error trace. Columns 3-6 represent the run-time in seconds, peak memory in MB, the number of solutions found as well as if the actual error inserted is found, respectively. Terms TO (MEMOUT) refers to a time-out (memory-out) being declared. For time-out conditions, partial results are listed in the table. For memory-out conditions, results for fully completed iterations are listed, however iterations that caused a memory-out are not counted since the CNF cannot be generated to obtain the results.

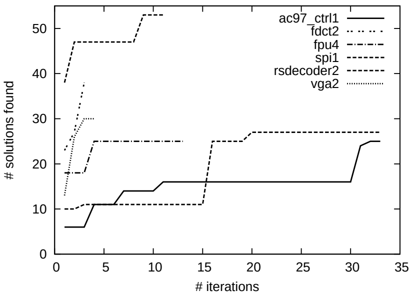


Fig. 9. # solutions vs. window expansion iterations

Finally, the last five columns of Table II show the results of the proposed window expansion technique. Algorithm 1 is implemented and run with $stride = 10$ until either the algorithm terminates or a time-out or memory-out is declared. A $stride$ of 10 is chosen so that each iteration of the window expansion would grow incrementally as to not cause an immediate memory-out. Columns 7-11 show the run-time in seconds, peak memory in MB, number of solutions returned, number of iterations run, and the iteration the actual error is found, respectively. A value of 0 in the last column translates to the error not being found by the algorithm.

The benefits of the window expansion technique are clear from the number of instances where it returns the error compared to a stand-alone debugger. Window expansion is able to find the actual error in 22 of the 28 instances whereas the stand-alone debugger is only able to find it in 6 cases due to 20 of the instances being declared with a memory-out due to excessive unrolling of the time-frames. This shows the value of incrementally formulating the debugging problem using window expansion so that the memory resources are not exhausted as in the case of just the stand-alone debugger.

This incremental formulation also allows the use of initial state suspects which are effective in 8 instances where complete results are declared before the entire error trace is analyzed. Moreover, in many of these instances, the actual error is found in the first iteration. This suggests that the empirical observation described in Section III is valid.

Figure 9 shows a better picture of how the excitation point of the error varies with the distance from the failure. The figure plots the number of solutions returned by window expansion against the number of iterations. Notice that after some point, the number of solutions returned by window expansion plateaus indicating that analyzing more of the error trace does not yield more suspects. For example, `fpu4` plateaus at 25 solutions in the fourth iteration and `spi1` plateaus at 27 in the twentieth iteration. This gives further evidence to confirm our empirical observation from Section III that errors are generally excited in close temporal proximity to their failure point.

TABLE II
BMD WITH WINDOW EXPANSION RESULTS

Instance Info		**Stand-alone debugger**				**BMD with Window Expansion**				
instance	# cycles	time (s)	mem (MB)	# sols	found	time (s)	mem (MB)	# sols	# iters	iter found
ac971	680	663	5557	34	yes	TO	3250	24	34	1
ac972	986	N/A	MEMOUT	0	no	TO	2621	41	30	30
div64bits1	110	TO	4614	0	no	TO	1783	30	4	2
div64bits2	110	TO	4299	0	no	TO	1468	18	4	2
fdct1	184	N/A	MEMOUT	0	no	TO	3985	41	3	2
fdct2	180	N/A	MEMOUT	0	no	TO	4194	37	4	1
fpu1	312	N/A	MEMOUT	0	no	TO	6291	15	15	1
fpu2	312	N/A	MEMOUT	0	no	31	557	4	1	1
fpu3	312	N/A	MEMOUT	0	no	TO	7025	8	17	4
fpu4	642	N/A	MEMOUT	0	no	TO	5872	30	14	1
fxu1	28	N/A	MEMOUT	0	no	N/A	MEMOUT	30	1	0
fxu2	28	N/A	MEMOUT	0	no	590	7655	24	1	1
fxu3	28	N/A	MEMOUT	0	no	TO	7550	40	1	1
mem_ctrl1	757	N/A	MEMOUT	0	no	27	409	11	1	1
mem_ctrl2	681	N/A	MEMOUT	0	no	26	437	5	1	1
rsdecoder1	115	TO	1153	138	yes	TO	644	141	4	0
rsdecoder2	196	2557	1783	53	yes	TO	1258	53	12	1
rx_comm1	253	N/A	MEMOUT	0	no	TO	4194	18	1	0
rx_comm2	156	N/A	MEMOUT	0	no	TO	4194	17	1	0
rx_comm3	99	N/A	MEMOUT	0	no	TO	4089	15	1	0
rx_comm4	195	N/A	MEMOUT	0	no	TO	7235	44	2	0
spi1	578	103	811	27	yes	TO	790	27	58	20
vga1	94	N/A	MEMOUT	0	no	103	1783	8	1	1
vga2	507	N/A	MEMOUT	0	no	1744	6291	29	5	3
vga3	861	N/A	MEMOUT	0	no	TO	7332	25	8	1
vga4	561	N/A	MEMOUT	0	no	TO	7961	30	8	4
wb1	132	5	221	7	yes	3	86	7	1	1
wb2	132	5	183	4	yes	2	83	4	1	1

B. *BMD with Window Partitioning*

The experimental results for the window partitioning technique are shown in Table III. Algorithm 2 is implemented and run with a fixed number of iterations ($r = \lceil k/step \rceil$) for each instance. As in Table II, each row corresponds to an instance of the designs in Table I with different values of r. The same notation for time-out and memory-out are also used. Again, partial results are available for the time-out case and in the memory-out case for iterations that fully complete.

The first column of Table III indicates the instance name. These are the same instances from Table II so the results are comparable. The rest of the columns are divided into three sets of experiments with r set to 2, 5 and 10 respectively. Columns 2-5 correspond to $r = 2$, 6-9 for $r = 5$ and 10-13 for $r = 10$. Each set of experiments shows the run-time in seconds, peak memory in MB, the number of solutions returned as well as if the actual error is found.

The benefit of the window partitioning technique is apparent when looking at the number of memory-out conditions compared with using the entire error trace. The number of memory-out cases are 15, 7 and 3 for $r = 2$, $r = 5$ and $r = 10$ respectively, compared with 20 for the full error trace. As expected the number of memory-out conditions decrease with smaller window sizes because the peak memory for the entire run will be decreased with the use interpolants.

From Table III, for the same instance, different values of r produce different results with respect to memory-out conditions. In some cases, larger values of r memory-out while smaller ones run successfully. This can be explained because the size of the interpolant is not necessarily proportional to the SAT instance that the proof of unsatisfiability is derived from. This causes the peak memory to grow larger at higher values of r. For example, `rsdecoder1` shows a case where at $r = 10$ a memory-out condition with peak memory greater than 8000 MB but $r = 2$ uses only 727 MB and $r = 5$ uses 3041 MB. However, the average memory reduction for $r = 2$, $r = 5$, and $r = 10$ is 19%, 41% and 49% respectively, compared to debugging the entire error trace, indicating that the general trend is still a decrease in the peak memory as r is increased.

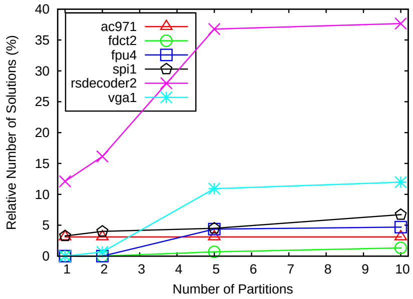


Fig. 10. Relative solutions vs. # partitions

A similar occurrence happens for run-time as well. For example, `fpu3` has a time-out at $r = 10$ while at $r = 5$ and $r = 2$ it is able to run to completion. This can be explained in two ways. First, at larger values of r there are more iterations which can potentially increase the run-time of the technique. Second, the run-time of the SAT instance is not necessarily proportional to the size of the instance causing additional discrepancy between the different values of r. Despite this variability, the reduction in run-time compared to a conventional debugger is still on average 39%, 53% and

TABLE III
BMD WITH WINDOW PARTITIONING RESULTS

Instance Info	**Interpolant, r=2**				**Interpolant, r=5**				**Interpolant, r=10**			
instance	time (s)	mem (MB)	# sols	found	time (s)	mem (MB)	# sols	found	time (s)	mem (MB)	# sols	found
ac971	399	2621	34	yes	177	1016	34	yes	141	554	34	yes
ac972	976	3985	49	yes	390	1573	52	yes	266	1049	67	yes
div64bits1	TO	1992	5	no	452	1049	32	yes	306	1049	47	yes
div64bits2	176	1992	20	yes	194	950	19	yes	279	856	47	yes
fdct1	N/A	MEMOUT	0	no	TO	4719	32	yes	1238	3251	62	yes
fdct2	N/A	MEMOUT	0	no	TO	4509	31	yes	N/A	MEMOUT	61	yes
fpu1	627	6921	16	yes	592	3880	16	yes	758	2412	31	yes
fpu2	275	6501	4	yes	108	2726	4	yes	61	1468	4	yes
fpu3	590	6501	8	yes	2806	3355	60	yes	TO	2726	30	yes
fpu4	N/A	MEMOUT	0	no	TO	5662	41	yes	1831	3355	44	yes
fxu1	N/A	MEMOUT	0	no	N/A	MEMOUT	30	yes	TO	7340	30	yes
fxu2	N/A	MEMOUT	0	no	TO	6711	26	yes	TO	6082	31	yes
fxu3	N/A	MEMOUT	0	no	TO	6187	46	yes	TO	5662	46	yes
mem_ctrl1	N/A	MEMOUT	0	no	486	6082	12	yes	206	2412	12	yes
mem_ctrl2	N/A	MEMOUT	0	no	180	4194	6	yes	92	2202	6	yes
rsdecoder1	TO	727	80	no	TO	3041	383	yes	N/A	MEMOUT	399	yes
rsdecoder2	1083	1678	65	yes	855	788	73	yes	TO	4928	109	yes
rx_comm1	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	TO	7979	120	yes
rx_comm2	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	TO	5453	128	yes
rx_comm3	N/A	MEMOUT	0	no	TO	7025	119	yes	TO	4404	150	yes
rx_comm4	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	TO	7550	127	yes
spi1	165	543	36	yes	324	715	82	yes	101	379	84	yes
vga1	490	6187	9	yes	1126	3146	146	yes	1058	2412	160	yes
vga2	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	TO	6921	160	yes
vga3	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	TO	5977	140	yes
vga4	N/A	MEMOUT	0	no	N/A	MEMOUT	0	no	N/A	MEMOUT	49	yes
wb1	4	131	7	yes	4	78	7	yes	4	88	7	yes
wb2	3	108	4	yes	3	76	4	yes	3	76	4	yes

59% for increasing values of r indicating that overall smaller windows still perform better.

The number of solutions returned by the window partitioning technique as r varies for several sample designs is shown in Figure 10. The solutions are calculated relative to the total number of potential suspects in the problem which are listed in Table I. A value of 0 indicates the problem did not return any results due to time-out or memory-out conditions. In this graph, $r = 1$ corresponds to the stand-alone debugger where the entire error trace is used. Notice that in most cases, the relative increase in suspects as r increases is relatively small. However, there can exist outliers such as rsdecoder2. In this instance, the interpolant is a weak constraint on the partition windows and results in a large increase in solutions. This outlines the trade-off when using window partitioning. In cases where the debugging problem requires excessive resources, the window partitioning technique effectively reduces the memory and run-time at the cost of a potential increase in the number of solutions.

C. Unified BMD Methodology

Figure 11 shows a comparison between a stand-alone debugger, window expansion and window partitioning with respect to number of instances solved. Each instance is categorized as either able to find the actual suspect, not able to find the suspect due to time-out, or not able to find the suspect due to memory-out. Both window expansion and window partitioning are able to find the error in more instances than a conventional debugger which is only able to find the actual error in 21% of the 28 cases. We can also see that the number of instances where the actual error is found also increases from 11 to 22 to 28 for increasing values of r. This occurs due to the fact that window partitioning is able to solve instances at larger r values without triggering a memory-out condition.

When compared to window expansion, window partitioning is able to find the actual error in 100% of the instances at $r = 10$ compared to window expansion finding the actual error in 79% of the 28 instances. While comparing the number of instances that yield complete results without a time-out or memory-out condition, $r = 10$ solves 14 instances while window expansion is only able to successfully complete 9 instances. This outlines one of the major benefits of window partitioning over window expansion showing that using interpolants can solve more instances while still maintaining complete results.

Comparing the results from Table II and Table III, the results show that both window expansion and window partitioning perform better in most instances than the full error trace with respect to run-time and peak memory. There are several outliers for both which occur for relatively small problems (less than 1000 seconds or 2 GB). This suggests for these smaller problems running a conventional debugger is sufficient and the BMD methodology may not be necessary. However for larger problems (greater 1000 seconds or 2 GB), the advantage of using BMD is clear as more instances are solved and dramatic reductions in run-time and peak memory are attained.

Table IV shows a separate set of experiments comparing the window expansion and window partitioning technique to determine the effect of the over-approximation of the window partitioning technique. Instances that are able to complete more than 10 iterations (100 time-frames) within the given

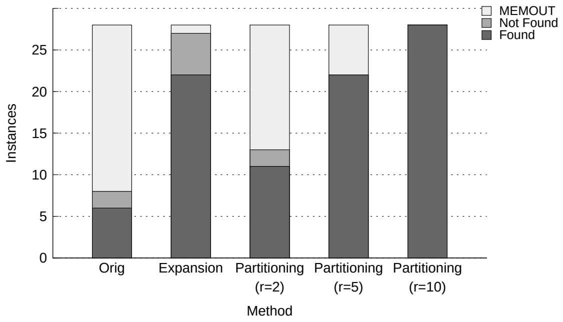


Fig. 11. Solved instances

TABLE IV
COMPARISON OF WINDOW EXPANSION AND WINDOW PARTITIONING WITH 100 TIME-FRAMES

Instance Info	Window Expansion (stride=50)			Window Partitioning (r=2)		
instance	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols
ac971	42.15	753	10	26.48	425	10
ac972	44.7	759	7	29.18	429	7
fpu1	320.04	4299	13	199.36	2202	15
fpu3	275.66	4194	8	127.87	2411	8
fpu4	417.35	4299	24	258.32	2307	30
rsdecoder2	168.77	773	52	146.92	586	56
spi1	11.39	150	10	10.26	101	26

time and memory limit are chosen for Table II. Each instance is modified so that the suffix of the error trace uses at most 100 time-frames. A stride of 50 is used for the window expansion technique and $r = 2$ for the window partitioning technique.

There is a small increase in the number of solutions across most instances such as `ac971` and `fpu4`. However in certain cases such as `spi1`, the increase in the number of solutions can be much larger (10 vs. 26) resulting in a great decrease in resolution. From the discussion in Section VI, this confirms the methodology of favouring the window expansion technique over the window partitioning technique because although on average the over-approximation may not cause a great decrease in resolution, there are certain cases in which there can be a large discrepancy. Since it is not known beforehand whether the specific instance is one of these cases, it is generally safer to use the window expansion technique if run-time and memory resources allow it.

Table V shows experiments run using the unified methodology. The hardest instances from Table III that cause a memory-out but are able to finish at least one iteration are used in these experiments. The columns of the table list the instance name, run-time, memory, number of solutions, additional solutions over Table III and if the actual error is found. The number of partitions ($r = \lceil k/step \rceil$) are listed beside the instance name and the $stride$ used is half of the corresponding $step$ value.

Benchmarks `fdct2`, `fxu1` and `rsdecoder1` all return more solutions than just using window partitioning alone. For instance, `fxu1` returned 57 solutions with the unified methodology compared with only 30 using window partitioning alone. The additional solutions that the algorithm is able to find are due to the fact that the window partitioning algorithm causes a memory-out during one of the iterations while the unified methodology did not. The memory-out condition is caused in the window partitioning experiments because the unrolled design with the interpolant is too large. However, in the case of the unified methodology the design is unrolled for fewer frames resulting in the ability to analyze more of the error trace. In the case of `vga4` however, fewer solutions are found because of the unified methodology took longer to run due to the increase in the number of iterations. Considering a longer time-out `vga4` would have found the same number of solutions before the memory-out condition because it occurred during the calculation of the interpolant which would be identical for both instances.

TABLE V
UNIFIED ALGORITHM RESULTS

instance	time (s)	mem (MB)	# sols	extra sols	found
fdct2 ($r = 10$)	TO	6920	62	1	yes
fxu1 ($r = 5$)	TO	7444	57	27	yes
rsdecoder1 ($r = 10$)	N/A	MEMOUT	404	5	yes
vga4 ($r = 10$)	TO	7759	48	-1	yes

VIII. CONCLUSION

Debugging today is a predominantly manual process that has become a bottleneck in the long and costly verification cycle. This work introduces the Bounded Model Debugging methodology to help conventional debuggers handle instances with long error traces. The first BMD technique is based on the empirical observation that the error is more likely to be excited within close temporal proximity to the failure point. It iteratively analyzes larger and larger bounded windows of the error trace until results are known to be complete. A second technique is also described in which the error trace is partitioned into non-overlapping windows each of which is separately analyzed. A set of comprehensive theoretical results are presented to guarantee the completeness of both approaches as well as an in-depth discussion of the two techniques. Based on those results, a unified methodology is later developed that leverages the strengths of both techniques. A set of extensive experiments on industrial designs show that the proposed framework finds the actual error in more than 79% of cases with the first BMD technique and 100% of cases with the second technique and only 21% of the cases without the BMD methodology. The proposed methodology allows large debugging problems with very long traces to be solved efficiently in terms of peak memory and run-time. The work presented here benefits existing debuggers as it allows them to handle real-life industrial problems.

IX. ACKNOWLEDGMENT

The authors would like to acknowledge the fundamental contribution of Prof. Farid Najm in the development of the probabilistic analysis in Section III.

REFERENCES

[1] D. McGrath, “De Geus touts new products, says ICs will rebound,” *EE Times*, March 2009.

[2] International Technology Roadmap for Semiconductors, “ITRS 2007,” http://www.itrs.net, 2009.

[3] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[5] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[6] R. Bryant, “Binary decision diagrams and beyond: Enabling techniques for formal verification,” in *Int’l Conf. on CAD*, 1995, pp. 236–243.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Design Automation Conf.*, 2001, pp. 530–535.

[8] A. Biere, “Resolve and expand,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2004, pp. 238–246.

[9] H. Konuk and T. Larrabee, “Explorations of sequential ATPG using Boolean satisfiability,” in *IEEE VLSI Test Symp.*, 1993, pp. 85–90.

[10] M. K. Ganai, A. Gupta, and P. Ashar, “Efficient SAT-based unbounded symbolic model checking using circuit cofactoring,” in *Int’l Conf. on CAD*, 2004, pp. 510–517.

[11] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, “Scalable automated verification via expert-system guided transformations,” in *Formal Methods in CAD*, 2004, pp. 159–173.

[12] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, “Solver technology for system-level to RTL equivalence checking,” in *Design, Automation and Test in Europe*, 2009.

[13] H. Foster, “Assertion-based verification: Industry myths to realities (invited tutorial),” in *Computer Aided Verification*, 2008, pp. 5–10.

[14] K.-H. Chang, V. Bertacco, and I. L. Markov, “Simulation-based bug trace minimization with BMC-based refinement,” in *ICCAD*, 2005, pp. 1045–1051.

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

[16] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah, “Improved design debugging using maximum satisfiability,” in *Formal Methods in CAD*, 2007.

[17] G. Fey, S. Staber, R. Bloem, and R. Drechsler, “Automatic fault localization for property checking,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 6, pp. 1138–1149, 2008.

[18] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2002.

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

[20] S. Safarpour, A. Veneris, and F. Najm, “Managing verification error traces with bounded model debugging,” in *ASP Design Automation Conf.*, 2010.

[21] B. Keng and A. Veneris, “Scaling VLSI design debugging with interpolation,” in *Formal Methods in CAD*, 2009.

[22] OpenCores.org, “http://www.opencores.org,” 2007.

[23] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[24] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, “Using unsatisfiable cores to debug multiple design errors,” in *Great Lakes Symp. VLSI*, 2008.

[25] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, “Backspace: Formal analysis for post-silicon debug,” in *Formal Methods in CAD*, 2008, pp. 1–10.

[26] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, “A formal approach for debugging arithmetic circuits,” in *IEEE Trans. on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.

[27] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem proving,” *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.

[28] L. Zhang, “Searching for truth: Techniques for satisfiability of Boolean formulas,” Ph.D. dissertation, Princeton, 2003.

[29] W. Craig, “Linear reasoning. a new form of the herbrand-gentzen theorem,” *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[30] K. McMillan, “Interpolation and SAT-based model checking,” in *Computer Aided Verification*, 2003.

[31] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, “Scalable don’t-care-based logic optimization and resynthesis,” in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 151–160.

[32] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

PLACE
PHOTO
HERE

Brian Keng received the B.A.Sc. degree in computer engineering from the University of Waterloo and the M.A.Sc. degree in computer engineering from the University of Toronto.

He is currently a Ph.D. student at the University of Toronto in the Department of Electrical and Computer Engineering. His research interests are in CAD for design debugging, test and verification of digital circuits and systems. In particular, he is interested in topics relating to the theory and application of formal methods.

PLACE
PHOTO
HERE

Sean Safarpour received the B.A.Sc. degree in computer engineering from the University of British Columbia, the M.A.Sc. and Ph.D. degrees in computer engineering from the University of Toronto.

He is currently chief technology officer at Vennsa Technologies, Toronto, Ontario, 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

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.