Debugging RTL using Structural Dominance

Hratch Mangassarian, *Member, IEEE*, Bao Le, *Student Member, IEEE*, and Andreas Veneris, *Senior Member, IEEE*

***Abstract*— RTL debug has become a resource-intensive bottleneck in modern VLSI CAD flows, consuming as much as 32% of the total verification effort. This work aims to advance the state-of-the-art in automated RTL debuggers, which return all potential bugs in the RTL, called *solutions*, along with corresponding *corrections*. First, an iterative algorithm is presented to compute the dominance relationships between RTL blocks. These relationships are leveraged to discover *implied* solutions with every new solution, thus significantly reducing the number of formal engine calls. Furthermore, a modern SAT solver is tailored to detect debugging *non-solutions*, sets of RTL blocks guaranteed to be bug-free, and imply other non-solutions using the precomputed RTL dominance relationships. Extensive experiments on industrial designs show a three-fold reduction in the number of SAT calls due to solution implications, coupled with faster SAT run-times due to non-solution implications, resulting in a 2.63x overall speed-up in total SAT solving time, demonstrating the robustness and practicality of the proposed approach.**

I. Introduction

With the growing size and complexity of VLSI designs, the disparity between our ability to design and to verify circuits, referred to as the *verification gap*, has become a major concern [1]. Today, the ratio of verification engineers to designers in the industry reaches 2:1 for complex designs [2] and has been projected to increase almost seven-fold by 2015 [3]. However, despite the allocation of extensive resources in an effort to bridge the verification gap, design verification consumes on average more than 50% of the VLSI design cycle [1].

Once functional verification discovers a discrepancy between a design and its specification, a *counter-example* is returned, consisting of a sequence of input stimuli that exhibits a mismatch between the actual and expected responses of the design and its specification, respectively. Given a buggy design and a counter-example, *design debugging* is the process of tracking down the root cause of the observed erroneous behavior. The latter is still a predominantly manual task in the industry, entailing the burdensome analysis of long and complex counter-examples [2]. Recent technical roadmaps and market studies suggest that once a design fails verification, debugging it and fixing it can consume up to 32% of the total verification effort [1].

H. Mangassarian is with Google Inc. This paper was written when he was with the ECE Department at the University of Toronto, Toronto, ON, Canada, M5S 3G4 (hratch.mangassarian@gmail.com).

B. Le is with the ECE Department at the University of Toronto, Toronto, ON, Canada, M5S 3G4 (lebao@eecg.toronto.edu).

A. Veneris is with the ECE and CS Departments at the University of Toronto, Toronto, ON, Canada, M5S 3G4 (veneris@eecg.toronto.edu).

With the aim of alleviating the design debugging cost, several methodologies have been proposed over the years to automate this process [4]–[16]. The output of a modern automated design debugger is a set of potential bug locations, referred to as *solutions*. Each solution denotes a set of RTL lines or blocks, where functional changes called *corrections* can rectify the erroneous behavior in the given counter-example. The automated debugger must return *all* solutions, along with their corrections, with engineers being given the final task of identifying the real bug and fixing it.

State-of-the-art automated debuggers make heavy use of formal tools, such as *Boolean satisfiability* (SAT) [11], *quantified boolean formulas* [13] and *maximum satisfiability* [17]. They reduce the debugging problem into a propositional formula whose satisfying assignments correspond to debugging solutions. As such, hundreds of formal engine calls are often required to return all solutions, one at a time [15]. With typical design sizes containing millions of synthesized gates and hundreds of thousands of RTL lines, the heavy computational cost of such a high number of formal engine calls limits the effectiveness and scalability of automated debugging software. This work proposes techniques that (a) reduce the number of required formal engine calls and (b) expedite the run-time of each call to the formal engine. This is done by leveraging structural dominance relationships between RTL components in the design.

A node u is said to be a *single-vertex dominator* of another node v if every path from v to a primary output passes through u. Single-vertex dominators can be found in linear-time [18], [19] and have been used for optimizing various CAD tasks, *e.g.*, test pattern generation [20], [21]. More recently, they have been leveraged in the gate-level debugger in [11], which performs an initial debugging pass on selected dominator gates. However, state-of-the-art automated design debuggers operate at the RTL-level [12], [14], where bugs occur in RTL *blocks* (*e.g.*, an *always* block, an *if* statement, a module definition, etc), corresponding to multiple-gate, multiple-output circuit blocks in the synthesized netlist. As such, it is difficult to make use of single-vertex dominators at the RTL-level. A block $\mathbf{a}$ dominates another block $\mathbf{b}$ if every path from every node in $\mathbf{b}$ to a primary output passes through a node in $\mathbf{a}$. In existing approaches for computing so-called *multiple-vertex* or *generalized* dominators, the gates constituting each block are not fixed in advance. Instead, nodes are grouped to form blocks *during the algorithm*, and according to certain conventions (*e.g.*, the smallest subset of fanouts collectively dominating a node [22], [23]). In contrast, we are interested in computing dominance relationships among blocks of nodes defined *a priori* by a hierarchical RTL design.

Our initial contribution is an algorithm that iteratively computes all the dominator RTL blocks of each RTL block in

the design. Next, we apply our algorithm as a preprocessing step to debugging, and leverage it in two ways.

First, we prove that for each solution RTL block returned by the automated debugger, blocks that dominate it are separate *implied* solutions. As such, the number of formal engine calls for finding all solutions can be significantly reduced using solution implications. Moreover, we show how to extract corrections for such implied solutions in linear time from the satisfying assignment corresponding to the original solution. This can be thought of as pruning the solution space of the debugging problem.

We also use block dominance to prune the non-solution space of the debugging problem. We introduce the concept of *non-solutions*, which are sets of blocks that cannot be modified in any way to correct the counter-example. We show that if a set of n RTL blocks is a non-solution, then a set of n blocks they dominate can also be ruled out as a non-solution. Detecting non-solutions and blocking the RTL blocks they dominate using blocking clauses during a SAT run can lead to significant time savings. In order to make such non-solution implications possible and useful, we present a new SAT branching scheme where *error-select* variables [11] are decided upon first, allowing the early detection of original non-solutions. We also prove that error-select variables are part of the *careset* [24] of the debugging problem, providing further theoretical ground for moving them up in the SAT decision tree. Finally, solution and non-solution implications are shown to be valid for any error cardinality.

The proposed techniques are presented and implemented on top of a SAT-based automated RTL debug framework [11], [12] using MINISAT 2.2.0 [25] as the back-end solver. An extensive set of experiments on real industrial designs obtained by our partners demonstrates the consistent benefits of the presented framework. It is shown that 66% of solutions are discovered early due to solution implications, resulting in a three-fold reduction in the average number of SAT solver calls. This, coupled with the fact that 25% of all non-solutions are implied and blocked, results in a 3x overall speed-up in solving time. These results demonstrate the effectiveness and practicality of our contributions.

The paper is organized as follows. Section II contains preliminaries on automated design debugging and dominators. Section III presents our iterative algorithm for computing dominance relationships between blocks and proves its correctness. Section IV shows how to leverage block dominators for on-the-fly solution implications in design debugging. Section V introduces debugging non-solutions and describes the use of block dominators to imply non-solutions. Section VI discloses the details of our tailored SAT solver, which can detect original non-solutions in order to imply further non-solutions based on block dominators. Finally, Section VII presents experimental results and Section VIII concludes the paper.

II. PRELIMINARIES

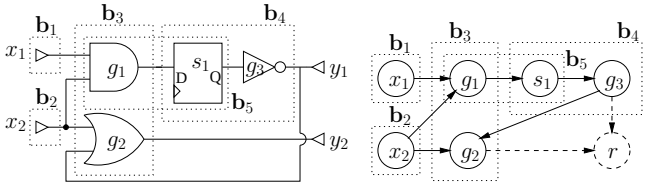
The following notation is used throughout the paper. Given a sequential circuit $\mathcal{C}$, the symbol $\mathbf{n}$ denotes the set of all nodes in $\mathcal{C}$. The symbols $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ label (possibly overlapping) subsets of $\mathbf{n}$, respectively referring to the sets of primary inputs, primary outputs and state elements (flip-flops) of $\mathcal{C}$. For each $\mathbf{z} \in \{\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{n}\}$, the Boolean variable z_i denotes the ith element in the set $\mathbf{z}$. In general, bold $(\mathbf{z})$ versus regular (z) symbols differentiate sets or sequences from single variables.

We consider designs with single clock domains, although the described theory is applicable to multiple synchronous clock domains using the techniques described in [26]. The authors of [26] show how to transform multiple synchronous clock domains into a single domain using a global, high-frequency clock and by adding extra circuitry around flip-flops and latches. The interested reader is referred to [26], as the details of this translation are beyond the scope of this paper.

Time-frame expansion for k clock-cycles is the process of replicating, or *unrolling*, the combinational component of $\mathcal{C}$ k times, such that the next-state of each time-frame is connected to the current-state of the next time-frame, thus modeling the sequential behavior of $\mathcal{C}$. For any variable (or set of variables) z_i (or $\mathbf{z}$), symbol z_i^t (or $\mathbf{z}^t$) denotes the corresponding variable (or set of variables) in time-frame t of the unrolled circuit. The behavior of $\mathcal{C}$ during the tth clock-cycle is formalized using the transition relation predicate $T(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t)$, which describes the dependence of the primary outputs $\mathbf{y}^t$ and next-state $\mathbf{s}^{t+1}$ on the primary inputs $\mathbf{x}^t$ and current-state $\mathbf{s}^t$. The transition relation T can be extracted from $\mathcal{C}$ and is normally given in Conjunctive Normal Form (CNF), using the set of nodes $\mathbf{n}^t$ as auxiliary variables.

An RTL design is translated into a gate-level netlist using logic synthesis. Such a gate-level sequential circuit $\mathcal{C}$ can also be represented as a directed graph. For convenience, we add an artificial sink node r to this graph, such that the set of nodes $V = \mathbf{n} \cup \{r\}$ and the set of edges $E = \{(n_i, n_j) | n_i$ is a fanin of n_j in $\mathcal{C}\} \cup \{(y_i, r) | \forall y_i \in y\}$. We reserve the letters u and v to refer to nodes in V. Let $fanout(v) = \{u \in V | (v, u) \in E\}$ and $fanin(v) = \{u \in V | (u, v) \in E\}$. Furthermore, the nodes $\mathbf{n}$ of $\mathcal{C}$ are grouped into (possibly overlapping) *blocks*. Each block consists of the synthesized gates of a given block of RTL code, such as an *always* block in Verilog. Let $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{|\mathcal{B}|}\}$ denote the set of all blocks, where each $\mathbf{b}_i \subseteq \mathbf{n}$ is a collection of nodes. Note that the same node v can belong to more than one block because of the hierarchical nature of RTL. The set $out(\mathbf{b}_i)$ denotes the outputs of block $\mathbf{b}_i$. In the unrolled circuit, the set $\mathbf{b}_i^t$ ($out(\mathbf{b}_i^t)$) contains the (output) nodes of block $\mathbf{b}_i$ in time-frame t. Finally, for each node v, we let $out^{-1}(v) = \{\mathbf{b}_j | v \in out(\mathbf{b}_j)\}$ denote the set of blocks in which v is an output.

Consider the sequential circuit in Figure 1(a). The blocks



(a) $\mathcal{C}$, $\mathcal{B}$ (b) Graph representation of $\mathcal{C}$, $\mathcal{B}$

Fig. 1. A sequential circuit with blocks

$\{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$ are shown in dotted boxes. We have $out(\mathbf{b}_1) = \{x_1\}$, $out(\mathbf{b}_3) = \{g_1, g_2\}$ and $out(\mathbf{b}_4) = \{g_3\}$. Furthermore, $out^{-1}(g_3) = \{\mathbf{b}_4\}$, $out^{-1}(s_1) = \{\mathbf{b}_5\}$, $out^{-1}(g_1) = \{\mathbf{b}_3\}$ and $out^{-1}(r) = \emptyset$. Note that y_1 and y_2 are primary output labels for g_3 and g_2, respectively, and do not represent separate nodes. Figure 1(b) presents the corresponding directed graph, including the artificial sink r.

A. Single-Vertex Dominators

In a directed graph $\mathcal{C} = (V, E, r)$ with a single output sink $r \in V$, a node $u \in V$ is said to be a structural single-vertex post-dominator, or simply *dominator*, of a node $v \in V$, if every path from v to the sink r passes through u. The set $dom(v) = \{u \in V | u \text{ dominates } v\}$ consists of nodes that dominate v. As a convention, we consider that a node dominates itself. Furthermore, to ease the presentation, we assume that every node has a path to r (*i.e.*, all dangling logic has been removed).

The *immediate dominator* of a node v ($v \neq r$), denoted by $idom(v)$, is a provably unique node u ($u \neq v$) that dominates v and is dominated by all the nodes in $dom(v) - \{v\}$. It can be shown that for all $v \in V - \{r\}$, $dom(v) = \{v\} \cup idom(v) \cup idom(idom(v)) \cup \cdots \cup \{r\}$ [27]. Therefore it is sufficient to compute all immediate dominators, which can be done in $O(|E| + |V|)$ time [18], [19]. In the directed graph shown in Figure 1(b), $dom(x_1) = \{x_1, g_1, s_1, g_3, r\}$, $dom(x_2) = \{x_2, r\}$, $idom(x_1) = \{g_1\}$, $idom(x_2) = \{r\}$.

In this work, we are interested in finding dominance relationships between blocks in $\mathcal{B}$, rather than between nodes in V. Section III outlines our approach, and discusses why methods for computing single-vertex dominators, as well as existing techniques for computing multiple-vertex dominators are not applicable in a design debugging setting.

B. Design Debugging

This section describes SAT-based design debugging [11] and introduces relevant notation, which is used throughout the paper. Given an erroneous design $\mathcal{C}$, a set of blocks $\mathcal{B}$, a counter-example of length k (along with its expected outputs) and an error cardinality N, the task of an automated design debugger is to find all sets of N blocks that can be responsible for the counter-example. More precisely, each returned set of N blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, where $\{i_1, \ldots, i_N\} \subseteq \{1, \ldots, |\mathcal{B}|\}$, can be modified to rectify the erroneous behavior exhibited in the counter-example. We refer to each such set of N blocks as a *solution* of cardinality N. SAT-based automated design debugging [11], [12] encodes the debugging problem as a propositional formula whose satisfying assignments correspond to debugging solutions. The following are the steps to translate design debugging into a SAT problem. We use $\mathcal{C}$ and $\mathcal{B}$ given in Figure 1(a) as an example for illustrating the encoding process. Figure 2 is an illustration of the resulting design debugging encoding for a two-cycle counter-example.

First, a set of *error-select* variables $\mathbf{e} = \{e_1, \ldots, e_{|\mathcal{B}|}\}$ are added to the circuit, such that setting $e_i = 1$ disconnects gates in $out(\mathbf{b}_i)$ from their fanins, making them free variables, whereas setting $e_i = 0$ does not modify the circuit. This can be achieved by inserting special multiplexers or switches at block outputs or by directly modifying the CNF of the transition relation. Next, this enhanced circuit is replicated using time-frame expansion for the length of the counter-example k, and such that for all time-frames t, outputs $out(\mathbf{b}_i^t)$ are controlled by the same error-select variable e_i. Figure 2 illustrates this, where each e_i is shown as an enable on the side of gates in $out(\mathbf{b}_i^t)$, across all time-frames t. This allows the SAT solver to modify the outputs of block $\mathbf{b}_i$ *across all time-frames* by setting $e_i = 1$ to "fix" any potential errors in $\mathbf{b}_i$.

Then, a set of constraints are applied to the initial state, the primary inputs and primary outputs in order to ensure that given the initial state $\Phi_S(\mathbf{s}^1)$ and primary input values $\Phi_X(\mathbf{x}^1, \ldots, \mathbf{x}^k)$ in the counter-example, the primary outputs yield their *expected* values $\Phi_Y(\mathbf{y}^1, \ldots, \mathbf{y}^k)$ given by the specifications. Φ_Y can also be expressed as a set of properties. Finally, an error cardinality constraint $\Phi_N(\mathbf{e})$ is added, setting $\sum_{i=1}^{|\mathcal{B}|} e_i$ to a pre-specified constant N. The resulting propositional formula is given by:

$$Debug = \bigwedge_{t=1}^{k} T_{en}(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t, \mathbf{e}) \wedge \Phi_S(\mathbf{s}^1) \wedge \Phi_X(\mathbf{x}^1, \ldots, \mathbf{x}^k) \wedge \Phi_Y(\mathbf{y}^1, \ldots, \mathbf{y}^k) \wedge \Phi_N(\mathbf{e}) \quad (1)$$

where $T_{en}(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t, \mathbf{e})$ refers to the transition relation predicate of the enhanced circuit at time-frame t.

Each assignment to $\mathbf{e} = \{e_1, \ldots, e_{|\mathcal{B}|}\}$ satisfying $Debug$ (1) corresponds to a debugging solution, and the SAT solver must find *all* such satisfying assignments to $\mathbf{e}$. This is normally done by iteratively blocking each satisfying assignment using a blocking clause and re-solving $Debug$ until the problem becomes unsatisfiable. In a satisfying assignment where some $e_i = 1$, the values of $out(\mathbf{b}_i^t)$ across all time-frames $t = 1 \ldots k$ represent a sequence of *corrections*, which would correct the erroneous behavior in the counter-example.

Example 1 *Consider the sequential circuit in Figure 1(a) to be a buggy implementation. We are also given a two-cycle counter-example with initial state* 0*, inputs* $\langle x_1, x_2 \rangle = \langle 1, 1 \rangle$ *in the first time-frame and* $\langle 0, 1 \rangle$ *in the second, as well as expected outputs* $\langle y_1, y_2 \rangle = \langle 1, 1 \rangle$ *and* $\langle 0, 0 \rangle$ *in the first and second time-frames. This yields a mismatch in the second time-frame at the output* y_2*, since the buggy circuit yields* $y_2^2 = 1$.

The corresponding design debugging formulation is illustrated in Figure 2. The constraints $\Phi_S = \bar{s}_1^1$, $\Phi_X = x_1^1 x_2^1 \bar{x}_1^2 x_2^2$

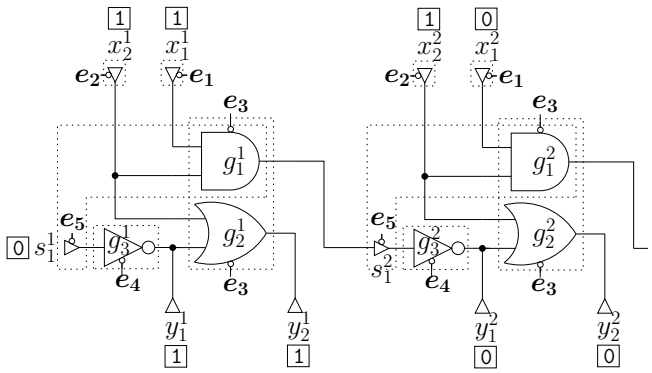


Fig. 2. Design debugging formulation

and $\Phi_Y = y_1^1 y_2^1 \bar{y}_1^2 \bar{y}_2^2$ are shown in boxes, while Φ_N is omitted for brevity. For $N = 1$, $\{\mathbf{b}_2\}$ and $\{\mathbf{b}_3\}$ will be returned by the solver as separate solutions, and are therefore considered potentially buggy blocks. Corrections for solution $\{\mathbf{b}_2\}$ (respectively $\{\mathbf{b}_3\}$) consist of the satisfying assignments to $\{x_2\}$ (respectively $\{g_1, g_2\}$) during the two time-frames. For instance, in any correction for solution $\{\mathbf{b}_2\}$, x_2^2 must be set to 0.

III. Dominance between Blocks

In this section, an iterative algorithm is presented for computing the dominator RTL blocks of every RTL block.

Definition 1 *A block $\mathbf{b}_j$ dominates another block $\mathbf{b}_i$, denoted as $\mathbf{b}_j \mathsf{D} \mathbf{b}_i$, if and only if every path from every node in $\mathbf{b}_i$ to a primary output in $\mathbf{y}$ passes through a node in $\mathbf{b}_j$.*

Assuming that internal (non-output) block nodes cannot be primary outputs, any path to a primary output exiting a block must pass through one of its outputs. Furthermore all primary outputs are connected to the artificial sink r. As such, the block dominator relation $\mathsf{D} \subseteq \mathcal{B} \times \mathcal{B}$ can be formalized using restricted quantifier notation [28] as follows:

$$\mathbf{b}_j \mathsf{D} \mathbf{b}_i \Leftrightarrow \forall v[v \in out(\mathbf{b}_i)].\forall \mathbf{p}[v \overset{\mathbf{p}}{\rightsquigarrow} r].\exists u[u \in \mathbf{p}].(u \in out(\mathbf{b}_j)) \quad (2)$$

where a path $\mathbf{p} : v \overset{\mathbf{p}}{\rightsquigarrow} r$ is a sequence of nodes starting at v and ending at r. The right-hand-side of Equation 2 reads *"for all vertices v in $out(\mathbf{b}_i)$, and for all paths $\mathbf{p}$ from v to r, there exists a vertex u in $\mathbf{p}$, such that $u \in out(\mathbf{b}_j)$"*.

We let the set $\mathsf{D}(\mathbf{b}_i) = \{\mathbf{b}_j | \mathbf{b}_j \mathsf{D} \mathbf{b}_i\}$ consist of blocks that dominate $\mathbf{b}_i$. Note that $\mathbf{b}_i \mathsf{D} \mathbf{b}_i$ according to (2). Consider the sequential circuit given in Figure 1(a). Although x_2 is not dominated by g_1 or g_2 separately, block $\mathbf{b}_2 = \{x_2\}$ is dominated by block $\mathbf{b}_3 = \{g_1, g_2\}$.

The relation D on the blocks $\mathcal{B}$ of $\mathcal{C}$ in Figure 1(b) is illustrated in Figure 3. Unlike single-vertex dominators, a block does not necessarily have a unique immediate dominator block. This can be seen for block $\mathbf{b}_1$ in Figure 3. As such, algorithms for calculating single-vertex immediate dominators cannot be used for computing block dominators. On the other hand, in existing approaches for computing so-called *generalized* or multiple-vertex dominators [22], [23], [29], block boundaries are not defined in advance. Instead, nodes are assembled into multiple-vertex dominators on-the-fly according to certain conventions, *e.g.*, the smallest subset of $fanout(v)$ collectively dominating a node v [22], [23]. This is not applicable in a design debugging setting, where circuit blocks are defined in advance by the hierarchical RTL design.

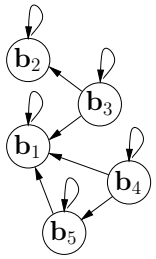


Fig. 3. Block dominator relation D of $\mathcal{C}$

Algorithm 1: Compute Block Dominators

input : Directed graph $\mathcal{C} = (V, E, r)$, blocks $\mathcal{B}$
output: Block dominator relation D

1 $V \leftarrow$ REVERSEPOSTORDERING($\mathcal{C}^T, r$);
2 // For each node v, compute $out^{-1}[v]$
3 **foreach** $v \in V$ **do** $out^{-1}[v] \leftarrow \emptyset$;
4 **foreach** $\mathbf{b}_i \in \mathbf{b}$ **do**
5 **foreach** $v \in out[\mathbf{b}_i]$ **do** $out^{-1}[v] \leftarrow out^{-1}[v] \cup \mathbf{b}_i$;
6 // Compute the relation d
7 $\mathsf{d}[r] \leftarrow \emptyset$;
8 **foreach** $v \in V - \{r\}$ **do** $\mathsf{d}[v] \leftarrow \mathcal{B}$;
9 $changed \leftarrow true$;
10 **while** $changed$ **do**
11 $changed \leftarrow false$;
12 **foreach** $u \in V$ *in reverse postorder* **do**
13 $blocks \leftarrow \bigcap_{\forall v \in fanout[u]} \left(\mathsf{d}[v] \cup out^{-1}[v]\right)$;
14 **if** $blocks \neq \mathsf{d}[u]$ **then**
15 $\mathsf{d}[u] \leftarrow blocks$;
16 $changed \leftarrow true$;
17 **foreach** $v \in V$ **do** $\mathsf{d}[v] \leftarrow \mathsf{d}[v] \cup out^{-1}[v]$;
18 // Compute the relation D
19 **foreach** $\mathbf{b}_i \in \mathcal{B}$ **do** $\mathsf{D}[\mathbf{b}_i] \leftarrow \bigcap_{\forall v \in out[\mathbf{b}_i]} \mathsf{d}[v]$;

In this work, the block dominator relation D on the set of blocks $\mathcal{B}$ is computed in two steps. First, the block dominators of each node $v \in V$ are computed. Then, these block-to-node dominators are used to compute the block-to-block dominator relation D.

Definition 2 *A block $\mathbf{b}_j$ dominates a node v, denoted as $\mathbf{b}_j \mathsf{d} v$, if and only if every path from v to a primary output in $\mathbf{y}$ passes through a node in $\mathbf{b}_j$.*

The block-to-node dominator relation $\mathsf{d} \subseteq \mathcal{B} \times V$ can be formalized as :

$$\mathbf{b}_j \mathsf{d} v \Leftrightarrow \forall \mathbf{p}[v \overset{\mathbf{p}}{\rightsquigarrow} r].\exists u[u \in \mathbf{p}].(u \in out(\mathbf{b}_j)) \quad (3)$$

We let the set $\mathsf{d}(v) = \{\mathbf{b}_j | \mathbf{b}_j \mathsf{d} v\}$ consist of blocks that dominate node v. For instance, in Figure 1(b), $\mathsf{d}(x_1) = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$, $\mathsf{d}(x_2) = \{\mathbf{b}_2, \mathbf{b}_3\}$ and $\mathsf{d}(s_1) = \{\mathbf{b}_4, \mathbf{b}_5\}$.

Algorithm 1 shows our pseudocode for computing the block dominator relation D. It first computes the sets $\mathsf{d}(v)$ for every $v \in V$ (lines 1 to 17). This is done using an iterative algorithm, where the set of block dominators of each node is initialized to all blocks $\mathcal{B}$ and iteratively refined until it converges to its actual block dominators. These block-to-node dominators are subsequently used on line 19 to compute $\mathsf{D}(\mathbf{b}_i)$ for every $\mathbf{b}_i \in \mathcal{B}$.

On line 1, $\mathcal{C}^T$ denotes the transpose of directed graph $\mathcal{C}$ (*i.e.*, $\mathcal{C}$ with edges reversed). The function REVERSEPOSTORDERING($\mathcal{C}^T, r$) performs a depth-first search (DFS) of $\mathcal{C}^T$ starting from r, and sorts the

nodes in *decreasing finishing times*. In general, a reverse postordering is not unique. For instance, for $\mathcal{C}$ given in Figure 1(b), REVERSEPOSTORDERING$(\mathcal{C}^T, r)$ can return $\langle r, g_2, g_3, s_1, g_1, x_2, x_1\rangle$. Traversing V in reverse postorder guarantees for each node $u \in V$ that at least one of $v \in fanout(u)$ is already visited by the time u is traversed. This will reduce the number of iterations needed for convergence when computing the sets $\mathsf{d}(v)$ later in the algorithm.

Lines 3 to 5 calculate the sets $out^{-1}(v)$ for each node v. The algorithm for computing the sets $\mathsf{d}(v)$ for all nodes v (lines 7 to 16) is based on the traditional data-flow analysis algorithm for finding single-vertex dominators [27], [30]. Lines 7 and 8 initialize each dominator set $\mathsf{d}(v)$ to all blocks $\mathcal{B}$ for $v \in V - \{r\}$, and to the empty set for $v = r$. In each iteration of the **while** loop, the nodes are traversed in reverse postorder (as calculated on line 1) and a refined set of dominator blocks is computed for each node on line 13. The computation of this refined set of dominator blocks of each node on line 13 is the main difference with the data-flow analysis algorithm for single-vertex dominators. The new set of dominator blocks of a node $u \in V$ is updated to be the intersection, over all $v \in fanout(u)$, of the dominator blocks of v as well as the blocks in which v is an output. If any of the sets $\mathsf{d}(v)$ are changed during an iteration (*i.e.,* the **if** condition on line 14 is true), the **while** loop is executed again. The **while** loop terminates after an iteration where all block-to-node dominator sets remain unchanged. Line 17 adds the blocks in which node v is an output, to the dominators of v. Finally, on line 19, the block dominators $\mathsf{D}(\mathbf{b}_i)$ of each block $\mathbf{b}_i$ are computed by intersecting the block dominators of each node in $out(\mathbf{b}_i)$.

Example 2 *We will go through Algorithm 1 for the graph respresentation of the circuit in Figure 1(a), along with its suspect blocks $\mathcal{B}$, as shown in Figure 1(b). Let the reverse postordering returned by* REVERSEPOSTORDERING$(\mathcal{C}^T, r)$ *be* $\langle r, g_2, g_3, s_1, g_1, x_2, x_1\rangle$. *After line 5, we have:*

$$out^{-1}(r) = \emptyset \quad out^{-1}(g_2) = \{\mathbf{b}_3\} \quad out^{-1}(g_3) = \{\mathbf{b}_4\}$$
$$out^{-1}(s_1) = \{\mathbf{b}_5\} \quad out^{-1}(g_1) = \{\mathbf{b}_3\} \quad out^{-1}(x_2) = \{\mathbf{b}_2\}$$
$$out^{-1}(x_1) = \{\mathbf{b}_1\}$$

After line 8, we have $\mathsf{d}(r) = \emptyset$ *and for every other* v, $\mathsf{d}(v) = \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$. *Next, each iteration of the* **while** *loop goes through every vertex* u *in reverse postorder and sets* $\mathsf{d}(u) = \bigcap_{\forall v \in fanout(u)} \big(\mathsf{d}(v) \cup out^{-1}(v)\big)$. *In the first iteration, the block-to-node dominators are set as follows:*

$$\mathsf{d}(r) = \emptyset$$
$$\mathsf{d}(g_2) = \mathsf{d}(g_3) = \mathsf{d}(r) \cup out^{-1}(r) = \emptyset$$
$$\mathsf{d}(s_1) = \mathsf{d}(g_3) \cup out^{-1}(g_3) = \{\mathbf{b}_4\}$$
$$\mathsf{d}(g_1) = \mathsf{d}(s_1) \cup out^{-1}(s_1) = \{\mathbf{b}_4, \mathbf{b}_5\}$$
$$\mathsf{d}(x_2) = \big(\mathsf{d}(g_1) \cup out^{-1}(g_1)\big) \cap \big(\mathsf{d}(g_2) \cup out^{-1}(g_2)\big) = \{\mathbf{b}_3\}$$
$$\mathsf{d}(x_1) = \mathsf{d}(g_1) \cup out^{-1}(g_1) = \{\mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$$

The second iteration of the **while** *loop does not modify any of these sets, and as such the loop is exited.*

Line 17 then adds $out^{-1}(v)$ *to every* $\mathsf{d}(v)$*:*

$$\mathsf{d}(r) = \emptyset \quad \mathsf{d}(g_2) = \{\mathbf{b}_3\} \quad \mathsf{d}(g_3) = \{\mathbf{b}_4\}$$
$$\mathsf{d}(s_1) = \{\mathbf{b}_4, \mathbf{b}_5\} \quad \mathsf{d}(g_1) = \{\mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$$
$$\mathsf{d}(x_2) = \{\mathbf{b}_2, \mathbf{b}_3\} \quad \mathsf{d}(x_1) = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$$

Finally, line 19 intersects the block-to-node dominators of the output nodes of each block to get the block-to-block dominator relation D *shown in Figure 3.*

Lemma 1 *The* **while** *loop in Algorithm 1 terminates and the block-to-node dominator relation* d *is correctly computed by the end of the* **foreach** *loop on line 17.*

Proof: In [31], the authors describe a class of well-known iterative *data-flow analysis* algorithms, which have a variety of applications (*e.g.,* in compiler optimization [32]) and are not restricted to calculating dominators. It can be shown that Algorithm 1 up to line 16 is a special case of Algorithm MK in [31]. Using the results in [31], one can show that the block-to-node dominator relation d is correctly computed by the end of the **foreach** loop on line 17. ■

Theorem 1 *Algorithm 1 correctly computes the block dominator relation* D.

Proof: $\mathsf{D}(\mathbf{b}_i)$ is computed on line 19 as $\bigcap_{\forall v \in out(\mathbf{b}_i)} \mathsf{d}(v)$. Using Lemma 1, we get:

$$\mathsf{D}(\mathbf{b}_i) = \bigcap_{\forall v \in out(\mathbf{b}_i)} \{\mathbf{b}_j | \forall \mathbf{p}[v \overset{\mathbf{p}}{\rightsquigarrow} r].\exists u[u \in \mathbf{p}].(u \in out(\mathbf{b}_j))\}$$
$$= \{\mathbf{b}_j | \forall v[v \in out(\mathbf{b}_i)].\forall \mathbf{p}[v \overset{\mathbf{p}}{\rightsquigarrow} r].\exists u[u \in \mathbf{p}].(u \in out(\mathbf{b}_j))\}$$

which satisfies the definition of the block dominator relation D given in (2). ■

The overall run-time of Algorithm 1 is normally dictated by the run-time of the **while** loop from line 10 to 16. Furthermore, during each iteration of the **while** loop, line 13 clearly dominates computation time. We assume that all dangling logic has been removed during preprocessing (*i.e.,* every node has a path to r), and as such $|V| = O(|E|)$. Using an aggregate analysis of all executions of line 13 during a single iteration of the **while** loop, it can be seen that line 13 performs a total of $O(|E|)$ intersections and unions between two sets of size at most $|\mathcal{B}|$ (since $\mathsf{d}(v), out^{-1}(v) \subseteq \mathcal{B}$). We assume that all sets are implemented using ordered lists and therefore intersections and unions can be done in linear time. As such, in a single iteration of the **while** loop, line 13 takes $O(|\mathcal{B}| \cdot |E|)$ time.

Let c denote the so-called *loop-connectedness* of the directed graph $\mathcal{C} = (V, E, r)$, which refers to the maximum number of *back edges* in any cycle-free path in $\mathcal{C}$. The back edges are defined according to the DFS performed in REVERSEPOSTORDERING$(\mathcal{C}^T, r)$ on line 1. It is proven in [31] that the number of iterations of the **while** loop for this class of algorithms is bounded by $c+2$. Hence, our algorithm takes $O(c \cdot |\mathcal{B}| \cdot |E|)$ time.

IV. Leveraging Block Dominance in Design Debugging: Solution Implications

In this section, we show how to leverage the relation D to imply debugging solutions. In effect, given a solution consisting of a set of blocks, we show that we can replace each block by any of its dominator blocks to get another solution. Formally, it is proven that for each known solution of $Debug$ (1) of the form $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, every set of the form $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ such that $\langle \mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N} \rangle \in \mathsf{D}(\mathbf{b}_{i_1}) \times \cdots \times \mathsf{D}(\mathbf{b}_{i_N})$ is also a solution of $Debug$. This is an implication of Theorem 2, which is more general because it is also used in Section V. Furthermore, Corollary 1 shows that corrections for each implied solution can be obtained automatically from the satisfying assignment of the original solution.

First, due to the fixed length of a given counter-example, we define the following, slightly modified concept of domination.

Definition 3 *We say that a block $\mathbf{b}_j$ dominates another block $\mathbf{b}_i$ within k cycles, denoted as $\mathbf{b}_j \mathsf{D}_\mathsf{k} \mathbf{b}_i$, if and only if every path containing at most k state elements, starting from every node in $out(\mathbf{b}_i)$ to r passes through a node in $out(\mathbf{b}_j)$.*

The following three Lemmas are used in the proof of Theorem 2 later in this section. In Lemma 2, $\bigcup_{m=1}^{n} \mathbf{b}_{i_m}$ (respectively $\bigcup_{m=1}^{n} \mathbf{b}_{j_m}$) denotes a "super-block" consisting of all nodes in blocks $\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}$ (respectively $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}$).

Lemma 2 *For any $n[1 \leq n \leq |\mathcal{B}|]$, $\bigwedge_{m=1}^{n}(\mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m}) \Rightarrow (\bigcup_{m=1}^{n} \mathbf{b}_{j_m})\, \mathsf{D}\, (\bigcup_{m=1}^{n} \mathbf{b}_{i_m})$*

Proof: If $\forall m[1 \leq m \leq n]$, any path from any node in $out(\mathbf{b}_{i_m})$ to a primary output passes through a node in $out(\mathbf{b}_{j_m})$, then clearly any path from any node in one of $out(\mathbf{b}_{i_1}), \ldots, out(\mathbf{b}_{i_n})$ to a primary output passes through a node in one of $out(\mathbf{b}_{j_1}), \ldots, out(\mathbf{b}_{j_n})$. ■

Lemma 3 *For any $k \geq 0$, $\mathbf{b}_j \mathsf{D} \mathbf{b}_i \Rightarrow \mathbf{b}_j \mathsf{D}_\mathsf{k} \mathbf{b}_i$*

Proof: If $\mathbf{b}_j \mathsf{D} \mathbf{b}_i$ then every path from $\mathbf{b}_i$ to a primary output passes through $\mathbf{b}_j$. In particular, all paths to a primary output with at most k state elements also pass through $\mathbf{b}_j$. ■

Lemma 4 *If $\mathbf{b}_j \mathsf{D}_\mathsf{k} \mathbf{b}_i$ in $\mathcal{C}$, then in the k-cycle time-frame expansion of $\mathcal{C}$, every path from every node in $out(\mathbf{b}_i^t)$ ($\forall t[1 \leq t \leq k]$) to any primary output in $\{y^t, \ldots, y^k\}$ passes through a node in $out(\mathbf{b}_j^{t'})$ (for some $t'[t \leq t' \leq k]$).*

Proof: True by construction. ■

We say that $n \leq N$ RTL blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}$ can be *extended* to a debugging solution of cardinality N if:

$$\left(Debug \wedge \bigwedge_{m=1}^{n} e_{i_m} \right) \text{ is } \mathsf{SAT}.$$

I.e., if there exist $N - n$ other blocks $\{\mathbf{b}_{i_{n+1}}, \ldots, \mathbf{b}_{i_N}\}$, such that $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ is a cardinality N solution.

Theorem 2 *Given an arbitrary set of $n \leq N$ RTL blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\} \subseteq \mathcal{B}$, if it can be extended to a debugging solution of cardinality N, then replacing these n blocks by n blocks that dominate them, $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}\} \subseteq \mathcal{B}$, can also be extended to a cardinality N solution.*

Formally, for any $n[1 \leq n \leq N]$:

$$\left[\left(Debug \wedge \bigwedge_{m=1}^{n} e_{i_m} \right) \textit{ is } \mathsf{SAT} \right] \wedge \bigwedge_{m=1}^{n} (\mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m}) \Rightarrow \left[\left(Debug \wedge \bigwedge_{m=1}^{n} e_{j_m} \right) \textit{ is } \mathsf{SAT} \right] \quad (4)$$

Proof: In what follows, we refer to the left-hand-side (respectively right-hand-side) SAT formula of the implication as the LHS (respectively RHS). Let π denote any satisfying assignment of the LHS $= Debug \wedge \bigwedge_{m=1}^{n} e_{i_m}$. Assuming that $\bigwedge_{m=1}^{n}(\mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m})$, we will construct an assignment π' satisfying the RHS $= Debug \wedge \bigwedge_{m=1}^{n} e_{j_m}$ to prove the claim.

Given any set of variables $\mathbf{z}$, we use the notation $\pi(\mathbf{z})$ (respectively $\pi'(\mathbf{z})$) to denote the truth assignment to $\mathbf{z}$ in the satisfying assignment π (respectively π'). Clearly, every variable in $\{e_{i_1}, \ldots, e_{i_n}\}$ must be set to 1 in $\pi(\mathbf{e})$. Furthermore, we let the remaining $N - n$ error-select variables that are set to 1 in $\pi(\mathbf{e})$ be denoted by $\{e_{i_{n+1}}, \ldots, e_{i_N}\}$. Using this notation, the cardinality-N solution of the LHS corresponding to π can simply be written as $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$. We refer to this set as $\mathbf{B}$.

We start by constructing $\pi'(\mathbf{e})$ from $\pi(\mathbf{e})$. This is equivalent to constructing a set of blocks $\mathbf{B}'$ that we will later show is a cardinality N solution of the RHS. Clearly, every variable in $\{e_{j_1}, \ldots, e_{j_n}\}$ must be set to 1 in $\pi'(\mathbf{e})$. Furthermore, for every $m = n+1, \ldots, N$:

(a) If $e_{i_m} \notin \{e_{j_1}, \ldots, e_{j_n}\}$, we let $\pi'(e_{i_m}) = 1$. In other terms, if the mth block in $\mathbf{B}$ is not already part of $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}\}$, add it to $\mathbf{B}'$.

(b) If $e_{i_m} \in \{e_{j_1}, \ldots, e_{j_n}\}$, then $\exists h \in \{1, \ldots, n\}$, such that $\mathbf{b}_{i_m} \mathsf{D} \mathbf{b}_{i_h}$, and we let $\pi'(e_{i_h}) = 1$. In other terms, if the mth block in $\mathbf{B}$ is one of $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}\}$, add the corresponding dominated block from within $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}$ to $\mathbf{B}'$.

We set every other error-select variable to 0 in $\pi'(\mathbf{e})$. The total number of error-select variables assigned to 1 in $\pi'(e)$ is exactly N, thus satisfying Φ_N.

Using the scheme given above, we have:

$$\mathbf{B}' = \{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}\} \cup \{\mathbf{b}_{i_{n+1}}, \ldots, \mathbf{b}_{i_N}\} \cup \text{a (possibly empty) subset of } \{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}. \quad (5)$$

It is easy to show that $\mathbf{B}' \mathsf{D} \mathbf{B}$. We are already given that for each $m = 1 \ldots n$, $\mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m}$. Each of the blocks in the other two subsets in $\mathbf{B}'$ shown in (5) already exists in $\mathbf{B}$ and therefore dominates itself by definition. As such, each block in $\mathbf{B}'$ dominates at least one block in $\mathbf{B}$, and therefore, by Lemma 2, $\mathbf{B}' \mathsf{D} \mathbf{B}$. Furthermore, by Lemma 3, we get $\mathbf{B}' \mathsf{D}_k \mathbf{B}$.

In the second half of this proof, we will use the fact that $\mathbf{B}' \mathsf{D}_k \mathbf{B}$ in order to construct a satisfying assignment π' for the RHS SAT formula from any satisfying assignment π of the LHS SAT formula. The assignment π' must set all the variables in $Debug$, which includes the k-time-frame expanded circuit obtained from $\mathcal{C}$ as described in Subsection II-B. Let $\mathcal{U}$ refer to this expanded circuit (*e.g.,* Figure 2). In what follows,

we refer to the blocks in $\mathbf{B}'$ as $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ in order to simplify notation. Let $I = \{\mathbf{b}_{i_n}^t | 1 \leq n \leq N, 1 \leq t \leq k\}$ (respectively $J = \{\mathbf{b}_{j_n}^t | 1 \leq n \leq N, 1 \leq t \leq k\}$) denote the union of all nodes in $\mathbf{B}$ (respectively $\mathbf{B}'$) across all time-frames in $\mathcal{U}$. Also, let $out(I)$ (respectively $out(J)$) refer to the set of outputs of I (respectively J). We will partition the nodes in $\mathcal{U}$ into three parts, $\mathcal{U}_I$, $\mathcal{U}_J$ and $\mathcal{U}_R$, as follows.

Let $\mathcal{U}_J$ denote the transitive fanout of $out(J)$ in $\mathcal{U}$. Let $\mathcal{U}_I$ denote the nodes in $\mathcal{U}$ that are in the transitive fanout of $out(I)$, but not in $\mathcal{U}_J$. Finally, let $\mathcal{U}_R$ consist of the remaining nodes in $\mathcal{U}$, outside $\mathcal{U}_I$ and $\mathcal{U}_J$. Using $\mathbf{B}'\mathsf{D}_k\mathbf{B}$ and Lemma 4, we can imply that any path from $out(I)$ to a primary output must pass through $out(J)$. As a result, these partitions of $\mathcal{U}$ can be represented by the diagram shown in Figure 4.

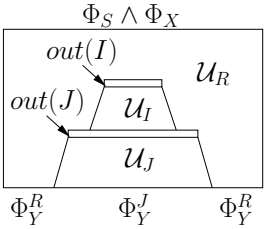


Fig. 4. Partition of $\mathcal{U}$

Note that in Figure 4, the output constraints are separated into two subsets: $\Phi_Y = \Phi_Y^J \wedge \Phi_Y^R$, where Φ_Y^J (respectively Φ_Y^R) denotes the output constraints applied at the outputs of $\mathcal{U}_J$ (respectively $\mathcal{U}_R$). This separation is only needed for this proof and is not required by our method.

We know that given $e_{i_1} = 1, \ldots, e_{i_N} = 1$, there exist assignments to the nodes in $\mathcal{U}_I, \mathcal{U}_J$ and $\mathcal{U}_R$ satisfying the LHS. Let $\pi(\mathcal{U}_I)$, $\pi(\mathcal{U}_J)$ and $\pi(\mathcal{U}_R)$ refer to these assignments. We want to find assignments $\pi'(\mathcal{U}_I)$, $\pi'(\mathcal{U}_J)$ and $\pi'(\mathcal{U}_R)$, such that given $e_{j_1} = 1, \ldots, e_{j_N} = 1$, the RHS is satisfied. These assignments are found as follows.

First consider the subset of output constraints applied at the outputs of $\mathcal{U}_R$, denoted by Φ_Y^R in Figure 4. Since $\pi(\mathcal{U}_R)$ satisfies Φ_Y^R and the input constraints to $\mathcal{U}_R$ (*i.e.*, $\Phi_S \wedge \Phi_X$) are the same in the LHS and the RHS, setting $\pi'(\mathcal{U}_R) = \pi(\mathcal{U}_R)$ will also satisfy Φ_Y^R in the RHS.

Next, consider $\mathcal{U}_I$. Note that any path from $out(I)$ to a primary output must pass through $out(J)$. Also, setting $e_{j_1} = 1, \ldots, e_{j_N} = 1$ in the RHS disconnects $out(J)$ from their fanins. Therefore, there are no output constraints applied on $\mathcal{U}_I$ (*i.e.*, $\mathcal{U}_I$ is dangling logic in the RHS). As such, $\pi'(\mathcal{U}_I)$ can simply "propagate" the values of $\pi'(\mathcal{U}_R)$ in $\mathcal{U}_I$.

Finally, since the nodes in $out(J)$ are disconnected from their fanins in the RHS, the SAT solver is free to pick any assignment for these variables. Furthermore, setting $\pi'(\mathcal{U}_R) = \pi(\mathcal{U}_R)$ already assigned any inputs to $\mathcal{U}_J$ coming from $\mathcal{U}_R$ to the same values as the LHS. Therefore, we can simply pick $\pi'(\mathcal{U}_J) = \pi(\mathcal{U}_J)$, which will satisfy Φ_Y^J in Figure 4. This completes the satisfying assignment π' to all the variables in $\mathcal{U}_I$, $\mathcal{U}_J$ and $\mathcal{U}_R$ in the RHS. Therefore, the RHS is SAT. ■

Corollary 1 *Given a solution $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ and its corresponding satisfying assignment π of $Debug$, a sequence of corrections for each implied solution $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ consists of the assignments to $\{out(\mathbf{b}_{j_n}^t) | 1 \leq n \leq N, 1 \leq t \leq k\}$ in π.*

Proof: Consider Theorem 2 with $n = N$. In its proof, we showed how to build a satisfying assignment π' of the RHS of (4) given a satisfying assignment π of the LHS. In particular, we showed that the subset of π' corresponding to $\mathcal{U}_J$ is the same as the subset of π corresponding to $\mathcal{U}_J$. In other terms, $\pi'(\mathcal{U}_J) = \pi(\mathcal{U}_J)$. Since $\mathcal{U}_J$ is simply the transitive fanout of $out(J)$ in $\mathcal{U}$, the subset of π' corresponding to $out(J)$ is also the same as the subset of π corresponding to $out(J)$. As such, given a satisfying assignment π for the original solution $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, a sequence of corrections for the implied solution $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ simply consists of the assignments in π to $out(J) = \{out(\mathbf{b}_{j_n}^t) | 1 \leq n \leq N, 1 \leq t \leq k\}$. ■

Intuitively, Theorem 2 and Corollary 1 show that if the SAT solver returns a given debugging solution, we can immediately imply that all its dominators are also solutions, and we can extract their corresponding corrections from the satisfying assignment of the original solution. This eliminates all the additional SAT solver calls to find these dominating solutions and their corrections, therefore significantly expediting the debugging process.

A. Debugging Flow using Solution Implications

The flowchart in Figure 5 illustrates the overall design debugging flow using on-the-fly solution implications. Algorithm 1 is first run to compute $\mathsf{D}(\mathbf{b}_i)$ for every block $\mathbf{b}_i \in \mathcal{B}$. Next, the automated debugger builds the CNF of $Debug$ and passes it to the SAT solver. If it is UNSAT, the flow terminates. Otherwise, a solution $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ is returned. A simple implication engine takes in this solution, and using the pre-computed block dominator relation D, generates all newly implied solutions. A blocking clause is added to $Debug$ for each of these implied solutions, as well as the original solution. The resulting debugging instance is given again to the automated debugger, and this process is repeated until the problem becomes UNSAT.

Example 3 *Consider the sequential circuit in Figure 1(a) and the corresponding design debugging formulation illustrated in Figure 2, with $N = 1$. Assume that $\mathsf{D} \subseteq \mathcal{B} \times \mathcal{B}$ has been computed using Algorithm 1. Let the solver first return the solution*

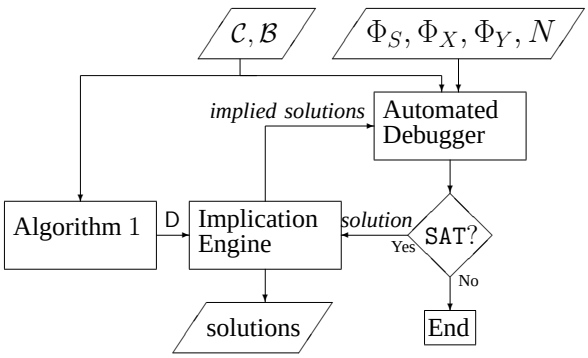


Fig. 5. Solution Implication Flow Chart

$\{\mathbf{b}_2\}$. *Since* $\mathsf{D}(\mathbf{b}_2) = \{\mathbf{b}_2, \mathbf{b}_3\}$*, the solution* $\{\mathbf{b}_3\}$ *(along with its corrections) can be immediately implied, eliminating a SAT call. After adding the corresponding blocking clauses* $(\bar{e}_2)$ *and* $(\bar{e}_3)$ *to* $Debug$*, the solver returns* UNSAT*, indicating that all solutions have been found.*

V. Leveraging Block Dominance in Design Debugging: Non-Solution Implications

In this section, we define the concept of a *non-solution*. We show that the reverse of the computed block dominance relationships can be leveraged to perform *non-solution implications*, thus pruning the search-space of the debugging problem. In Section VI, we present a tailored SAT solver which is able to learn and detect original non-solutions much faster, leading to earlier non-solution implications and expedited run-times.

As opposed to a solution, any set of N blocks whose outputs can in no way be simultaneously modified to correct the counter-example can be referred to as a non-solution. We extend this concept to sets of $n \leq N$ blocks as follows.

Definition 4 *Given an erroneous design* C*, a set of blocks* $\mathcal{B}$*, a counter-example of length* k *along with the corresponding expected outputs and an error cardinality* N*,* $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}$ *with* $n \leq N$ *is a* non-solution *if and only if* $Debug \wedge \bigwedge_{m=1}^{n} e_{i_m}$ *is* UNSAT*.*

In other terms, a set of $n \leq N$ blocks is an n-block non-solution if it cannot be extended to any solution of cardinality N. The following theorem proves that sets of blocks dominated by non-solutions are also non-solutions.

Theorem 3 *Given an erroneous design* $\mathcal{C}$*, a set of blocks* $\mathcal{B}$*, a counter-example of length* k *along with the corresponding expected outputs and an error cardinality* N*, if* $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_n}\}$ *with* $n \leq N$ *is a non-solution of* $Debug$ *and* $\bigwedge_{m=1}^{n} \mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m}$*, then* $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}$ *is also a non-solution.*

Proof: Let:

$$\Phi_i = \quad Debug \wedge \bigwedge_{m=1}^{n} e_{i_m} \qquad \Phi_j = \quad Debug \wedge \bigwedge_{m=1}^{n} e_{j_m}$$

Using Theorem 2, we have:

$$\left((\Phi_i \ \mathsf{SAT}) \wedge \bigwedge_{m=1}^{n} \mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m} \right) \Rightarrow (\Phi_j \ \mathsf{SAT})$$

$$\Leftrightarrow \quad (\Phi_i \ \mathsf{UNSAT}) \vee \neg \left(\bigwedge_{m=1}^{n} \mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m} \right) \vee (\Phi_j \ \mathsf{SAT})$$

$$\Leftrightarrow \quad (\Phi_i \ \mathsf{UNSAT}) \Leftarrow \left(\bigwedge_{m=1}^{n} \mathbf{b}_{j_m} \mathsf{D} \mathbf{b}_{i_m} \wedge (\Phi_j \ \mathsf{UNSAT}) \right)$$

■

Intuitively, Theorem 3 shows that if we are able to identify a non-solution, *i.e.*, a set of blocks that we cannot modify in any way to fix the given counter-example, we can immediately imply that all blocks dominated by this non-solution are also non-solutions. This information can be procured to the SAT solver by adding blocking clauses to divert it from considering these dominated blocks. This prunes its search space and therefore speeds-up the completion time of the SAT call.

Example 4 *Consider the sequential circuit in Figure 1(a) and the corresponding design debugging formulation illustrated in Figure 2, using* $N = 1$*. We know that block* $\mathbf{b}_4$ *dominates* $\mathbf{b}_1$ *and* $\mathbf{b}_5$*. If* $\mathbf{b}_4$ *is known to be a non-solution, using Theorem 3, we can imply that* $\mathbf{b}_1$ *and* $\mathbf{b}_5$ *are each separate non-solutions. We can therefore automatically add the clauses* $(\bar{e}_1)$ *and* $(\bar{e}_5)$ *to prune the search space of* $Debug$*.*

Whereas Theorem 2 can be used to imply solutions of cardinality N given each returned solution by the SAT solver, in order to be able to use Theorem 3 one must first be able to detect non-solutions. This can only be made possible by modifying the SAT solver.

One way to identify non-solution blocks is by monitoring learned clauses of the form $(\bar{e}_1 \vee \cdots \vee \bar{e}_n)$. However the SAT solver rarely learns such clauses. Instead, learned clauses are more complex and usually involve many other variables along with the error-select variables. Another way to detect blocks that are single-block non-solutions is to examine the forced assignments of *Boolean constraint propagation* (BCP) after each solver restart (when the decision stack is empty). If some $e_i = \mathtt{0}$ by unit propagation given an empty decision stack, then the solver has "learned" that $\mathtt{b}_i$ is a non-solution block. However, from our experiments, virtually all such non-solution blocks are learned during the last solver restart in the last call to the all-solution SAT procedure (after all solutions have been found), leaving little room for improvements using non-solution implications.

VI. A Tailored SAT Solver

In this section, we describe a new SAT branching scheme for design debugging, where error-select variables are decided upon first. This allows the early learning (and simple detection) of non-solutions, making non-solution implications useful.

A. Our SAT Branching Scheme

We force the SAT solver to first decide on all error-select variables ($\mathbf{e}$). The rest of this subsection provides several motivations for this choice. Furthermore, we force the solver to always assign error-select variables that are decided (*i.e.*, not forced due to BCP) to 1 before trying to set them to 0. The reason for doing this is to learn and detect non-solutions, and is explained in detail in Subsection VI-B. Once all the error-select variables are assigned, the solver uses the standard decision heuristics (*e.g.*, VSIDS [33]) for the remaining variables.

Motivation: A SAT solver can assign variables in any order. The first motivation for assigning the error-select variables early in the decision tree relates to their importance and their impact on other variable decisions in the SAT solving process. For example, when $e_i = \mathtt{1}$, the internal nodes of block $\mathbf{b}_i$ become dangling, and therefore they are don't-cares. As such, assigning the nodes in $\mathbf{b}_i$, as well as their fanouts, is useless if e_i is later assigned to 1.

What follows formalizes this motivation by proving that the error-select variables are part of the *careset* [24] variables of a design debugging problem. According to [24], a complete assignment on careset variables is a "gateway" to a satisfying assignment, and branching on these variables first can speed up SAT solvers by an order of magnitude.

In [24], a *constrained circuit* is one where certain variables are constrained to Boolean values. For instance, the constraint circuit corresponding to $Debug$ for the circuit in Figure 1(a) is shown in Figure 2. Given a constrained circuit and its corresponding CNF formula Φ, a partial assignment π is said to be a *minimally satisfying assignment* (MSA) if and only if:

(a) BCP cannot be applied on $\Phi|_\pi$ further.
(b) $\Phi|_\pi$ can be shown to be satisfiable by assigning all remaining unassigned inputs in its constrained circuit to *arbitrary* values and using BCP.
(c) Unassigning at least one variable in π would violate (a) or (b).

Here, $\Phi|_\pi$ denotes Φ where the variables in π have been assigned to their values in π.

Definition 5 *A non-empty set S of variables is a* careset *of Φ if every variable in S is assigned in every MSA of Φ.*

Theorem 4 *The error-select variables* $\mathbf{e}$ *belong to the* careset *of $Debug$.*

Proof: Assume towards a contradiction that a certain error-select variable e_i does not belong to the careset of $Debug$. Then there must exist an MSA π of $Debug$ where e_i is unassigned. By (b), $\Phi|_\pi$ must be satisfiable under any combination of assignments to the error-select variables not in π, including $e_i = \mathtt{0}$ and $e_i = \mathtt{1}$. This cannot be true because assigning e_i to $\mathtt{0}$ or $\mathtt{1}$ in a certain combination of error-select assignments will change the number of error-select variables set to $\mathtt{1}$, thus violating the error cardinality constraint Φ_N in at least one of the cases. ■

The second, and more important, reason for assigning the error-select variables early is that it allows the solver to learn non-solution blocks much faster. This in turn enables non-solution implications in order to prune the SAT search space earlier and therefore more effectively. Subsection VI-B discusses how to detect learned non-solutions using our branching scheme.

B. Detecting Non-Solutions

To simplify the presentation of this subsection, let us assume without loss of generality that the error-select variables, which are branched upon first in our SAT solver, are decided in the order of $\langle e_1, \ldots, e_{|B|}\rangle$. According to our branching scheme explained in Subsection VI-A, the SAT solver first assigns $e_1 = \mathtt{1}$. If the solver later switches to $e_1 = \mathtt{0}$ without finding a satisfying assignment under $e_1 = 1$, this means that $e_1 = \mathtt{1}$ cannot be extended to a satisfying assignment. Hence, $e_1 = \mathtt{0}$ is true for all satisfying assignments (if any exist). *I.e.,* $(\bar{e}_1)$ has been learned and $\{\mathbf{b}_1\}$ is a single-block non-solution.

The following Lemma shows that, using our decision scheme, assigning any e_i to $\mathtt{0}$ on the rightmost path in the SAT decision tree indicates that $\{\mathbf{b}_i\}$ is a single-block non-solution.

Lemma 5 *For any i, if every error-select variable in $\{e_1, \ldots, e_i\}$ is set to $\mathtt{0}$ by the SAT solver, then $\{\mathbf{b}_i\}$ is a single block non-solution.*

Proof: Recall that our decision scheme forces the solver to first set each error-select variable to $\mathtt{1}$ before trying to set it to $\mathtt{0}$. Thus, by construction, $e_i = \mathtt{1}$ has already been tried under every assignment combination to $e_1, \ldots, e_{i-1}$, and cannot be extended to a satisfying assignment. This means that $Debug \wedge e_i$ is $\mathtt{UNSAT}$, which implies that $\{\mathbf{b}_i\}$ is a single-block non-solution. ■

Example 5 *Consider the partial SAT decision tree in Figure 6(a), where error-select lines do not correspond to the blocks in Figure 1(a). Each set of blocks shown under a dashed line indicates a non-solution detected by the solver using Lemma 5. For instance, as soon as e_2 is assigned to $\mathtt{0}$, since its only ancestor e_1 is also assigned to $\mathtt{0}$, $\{\mathbf{b}_2\}$ is detected as a 1-block non-solution. The solver implies that every block that $\mathbf{b}_2$ dominates is also a non-solution, by Theorem 3 with $n = 1$. As such, we can add clauses of the form $(\bar{e}_i)$ for every block $\mathbf{b}_i$ that has $\mathbf{b}_2$ in its dominators $\mathsf{D}(\mathbf{b}_i)$.*

1) The General Case. Detecting n-Block Non-Solutions:

Theorem 5 *For any $n \leq N$ and for any $i \geq n$, if e_i is set to $\mathtt{0}$, and every error-select variable in $\{e_1, \ldots, e_{i-1}\}$ is set to $\mathtt{0}$ by the SAT solver except $n-1$ of them, say $\{e_{j_1}, \ldots, e_{j_{n-1}}\}$ which are set to $\mathtt{1}$, then $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_{n-1}}, \mathbf{b}_i\}$ is an n-block non-solution.*

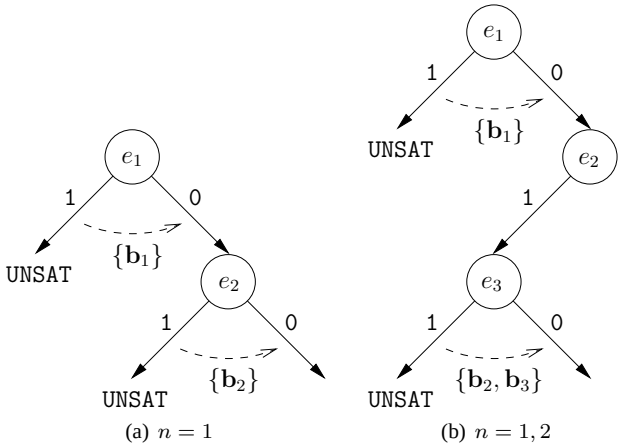
Proof: Once again, our decision scheme forces the solver to first set each error-select variable to $\mathtt{1}$ before trying to set it to $\mathtt{0}$. This means that every time the solver sets an error-select variable e_j to $\mathtt{0}$, it does so because it has exhausted the $e_j = \mathtt{1}$ branch and hence $e_j = \mathtt{0}$ is *implied* under the partial assignment to $e_1, \ldots, e_{j-1}$ above e_j in the decision tree.

As such, starting from the first error-select variable that is set to $\mathtt{0}$, which is implied by any previous error-select variable assignments to $\mathtt{1}$ (if any), it is easy to show by induction that every error-select variable assignment to $\mathtt{0}$ can be implied from any previous error-select variable assignments to $\mathtt{1}$. In other terms:

$$
\begin{aligned}
& e_i = \mathtt{0} \text{ is forced due to } e_{j_1} = \mathtt{1}, \ldots, e_{j_{n-1}} = \mathtt{1} \\
\Leftrightarrow \quad & Debug \rightarrow ((e_{j_1} \wedge \cdots \wedge e_{j_{n-1}}) \rightarrow \bar{e}_i) \\
\Leftrightarrow \quad & Debug \rightarrow (\bar{e}_{j_1} \vee \cdots \vee \bar{e}_{j_{n-1}} \vee \bar{e}_i) \\
\Leftrightarrow \quad & Debug \wedge (e_{j_1} \wedge \cdots \wedge e_{j_{n-1}} \wedge e_i) \text{ is } \mathtt{UNSAT} \\
\Leftrightarrow \quad & \{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_{n-1}}, \mathbf{b}_i\} \text{ is an } n\text{-block non-solution.}
\end{aligned}
$$

■

Example 6 *The partial SAT decision tree in Figure 6(b) shows non-solutions of one and two blocks, detected using*



(a) $n = 1$ (b) $n = 1, 2$

Fig. 6. Detecting n-block nonsolutions

Theorem 5. For instance, as soon as e_3 is assigned to 0*, since its only ancestor assigned to* 1 *is e_2 , we know that $\{\mathbf{b}_2, \mathbf{b}_3\}$ is a 2-block non-solution. Each of the blocks in this non-solution can be replaced by a block it dominates to get another non-solution, by Theorem 3 with $n = 2$. As such, we can add clauses of the form $(\bar{e}_i \vee \bar{e}_j)$ for every $\mathbf{b}_i$ and $\mathbf{b}_j$ dominated by $\mathbf{b}_2$ and $\mathbf{b}_3$, respectively.*

C. Restart Heuristic

Modern SAT solvers have periodic *restarts*, usually occurring after a certain number of conflicts. A restart clears assignments of all variables (including all decisions) while keeping the learned clauses. We modify the existing restart mechanism to enhance non-solution detection as follows. If no non-solutions have been learned during a solver restart, we generate a random number r ($1 \leq r \leq |B|$), and swap the order of the error-select variables below and above level r in the decision tree. The reasoning behind this is to avoid spending too much time in parts of the search-space where it is hard to detect and therefore imply non-solutions.

D. Debugging Flow using Solution and Non-Solution Implications

The flowchart in Figure 7 illustrates the overall design debugging flow using both solution and non-solution implications. This is an extension to the flow in Figure 5. Our

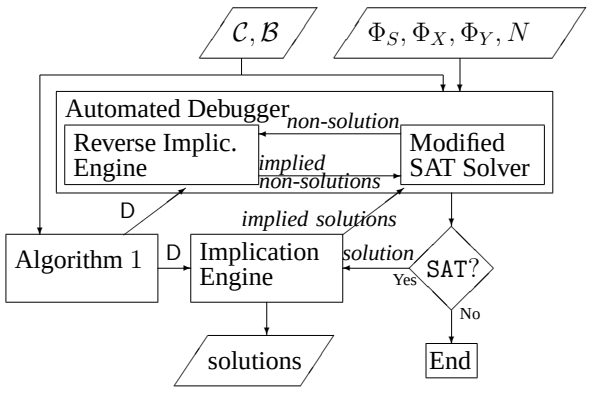


Fig. 7. Solution and Non-Solution Implication Flow Chart

modified SAT solver is able to detect n-block non-solutions, where $1 \leq n \leq N$, of the form $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_n}\}$. Each such non-solution is immediately passed to a reverse implication engine, which uses the pre-computed block dominator relation D to generate all implied non-solutions. These are passed back to the SAT solver, which adds the corresponding clauses to prune the search-space of the problem.

Example 7 *Consider the sequential circuit in Figure 1(a) and the corresponding design debugging formulation illustrated in Figure 2. Assume that $N = 1$, and that the modified SAT solver detects the non-solution $\{\mathbf{b}_4\}$. Since $\mathbf{b}_4 \in \mathsf{D}(\mathbf{b}_1)$ and $\mathbf{b}_4 \in \mathsf{D}(\mathbf{b}_5)$, the non-solutions $\{\mathbf{b}_1\}$ and $\{\mathbf{b}_5\}$ can be implied by the reverse implication engine. As such, the clauses $(\bar{e}_1)$ and $(\bar{e}_5)$ are added to $Debug$, immediately pruning the solution search-space.*

VII. Experimental Results

This section presents the experimental results for solution and non-solution implication-based design debugging in an industrial setting. All experiments are run using a single core of an i5-2400 3.1 GHz workstation with 8 GB of RAM and a timeout of 3600 seconds. The proposed debugging framework is implemented on top of a state-of-the-art SAT-based debugger based on [11], [12], [15], with a Verilog front-end to allow for RTL diagnosis. For non-solution implications, we tailor the debugger's back-end SAT solver, MINISAT-V2.2.0 [25].

Eight industrial Verilog designs from OpenCores [34] and two commercial designs provided by our industrial partners are used in our experiments. For each design, several debugging instances are generated by inserting different errors into the design. The RTL errors that are injected are based on the experience of our industrial partners. These are common designer mistakes such as wrong state transitions, incorrect operators or incorrect module instantiations [15]. Such errors typically translate to multiple gate-level changes. The erroneous design is then run through an industrial simulator with the accompanying testbench, where a failure is detected and a counter-example is recorded. Each block $\mathbf{b}_i \in \mathcal{B}$ consists of the synthesized gates corresponding to a (set of) line(s) in the RTL implementing an assignment, an if statement, a module definition, an instantiation, etc. Our RTL debugging tool uses *bounded model debugging* (BMD) [15], which is an orthogonal debugging optimization that helps deal with long counter-examples. BMD first examines the last d time-frames ($d = 40$ in this paper) of the counter-example, and iteratively analyzes earlier time-frames until all solutions are found. The reasoning behind it is that the bug is most often (although not always) excited within a few time-frames of it being observed. In our experiments, **trad** refers to the "traditional" debugging flow (without solution or non-solution implications), $+$**impl** includes only solution implications as illustrated in Figure 5, $-$**impl** includes only non-solution implications, and finally $+$**impl**$-$**impl** includes both as shown in Figure 7.

Table I presents the characteristics of each design debugging instance, as well as the number of solutions and the runtimes of **trad** with $N = 1$. The first column gives the

TABLE I
TRADITIONAL RTL DEBUG, $N = 1$

Instance	$\|\mathbf{n}\|$	$\|\mathcal{B}\|$	k	# sol	overh (s)	**trad** SAT (s)
vga-1	89402	1741	423	77	4.6	14.3
vga-2	89402	1741	11308	23	11.7	216
vga-3	89412	1593	465	21	10.4	42.7
vga-4	89412	1593	465	30	33.9	67.2
vga-5	89626	1534	34	46	55.7	110.2
ddr2-1	58399	2779	27	373	6.5	96.5
ddr2-2	64915	2777	27	509	6.9	90.2
ddr2-3	64915	2777	29	392	7.1	88.4
ddr2-4	58399	2779	29	373	6.6	78.5
dma-1	299862	8460	28	526	56.2	434.8
dma-2	191386	7874	35	468	15.6	196.4
dma-3	191397	6354	28	379	12.7	110.3
dma-4	191375	7479	27	227	11.8	65.5
dma-5	299838	8460	7	205	19.3	72.6
dma-6	301812	8460	20	134	41.0	59.8
mips789-1	63417	2747	31	241	12.7	109.7
mips789-2	63241	2580	22	166	20.5	117.6
mips789-3	63241	2750	24	162	10.2	40.8
usb-1	35158	3397	32	422	6.9	96.2
usb-2	35350	3401	106	322	24.6	519.5
usb-3	36180	3477	18	165	4.7	20.3
usb-4	35326	3118	56	157	7.2	16.5
usb-5	38346	3555	27	132	6.2	19.8
usb-6	39183	4389	21	151	6.2	28.8
rsdecoder-1	13564	2044	112	396	3.1	60.3
rsdecoder-2	13978	2044	142	308	4.1	T/O
hpdmc-1	11805	960	50	216	2.1	11.0
hpdmc-2	13181	1135	42	176	2.1	8.5
hpdmc-3	11547	1079	44	171	1.5	17.3
hpdmc-4	12849	1135	25	152	1.2	8.5
hpdmc-5	11548	1080	53	175	1.0	15.9
hpdmc-6	12834	1129	22	135	1.1	6.2
mrisc-1	17568	812	40	34	5.5	11.6
mrisc-2	18052	968	41	36	5.8	16.2
mrisc-3	17526	958	40	34	5.6	14.6
design1-1	690766	11738	27	241	134.8	1960.7
design1-2	45632	9156	29	166	4.8	112.6
design1-3	203718	10258	151	127	28.9	114.4
design2-1	875837	2592	132	99	13.5	11.8
design2-2	875837	2592	638	87	36.4	6.0
design2-3	875837	2863	338	117	27.1	19.7

instance name, which consists of the design name and an appended number indicating a different inserted error. The following three columns respectively show the number of circuit nodes $|\mathbf{n}|$ in the cone-of-influence of the counter-example, the number of blocks $|\mathcal{B}|$, the number of clock-cycles k in the counter-example. Columns *# sols* and *overh* respectively refer to the total number of returned solutions, and the run-time overhead in seconds for setting up the problem (*i.e.*, generating the CNF of $Debug$). This overhead includes graph optimizations such as dangling logic removal. Note that the *# sols* and *overh* numbers are common to all debugging flows. Finally, column **trad** *SAT* gives the total SAT solver run-time for returning all solutions with the traditional debugging flow. This is the sum of all SAT calls, including the last call which yields UNSAT.

Table II presents the experimental results for +**impl**, −**impl** and +**impl**−**impl**, for the same debugging instances as the ones shown in Table I. Column *Alg. 1* gives the run-time in seconds of Algorithm 1 for computing the block dominator relation D, which is run as a preprocesing step in each of these three flows. Next, for each of +**impl**, −**impl** and +**impl**−**impl**, column *SAT* shows the SAT solver run-time in seconds, whereas columns *impr* and *impr-o* show the speed-up achieved by that debugging method over **trad**, where *impr* excludes and *impr-o* includes the common overhead shown under column *overh* in Table I. The run-time of Algorithm 1 is included in speed-up calculations. Finally, under +**impl**−**impl**, columns *sol impl* and *non-sol impl* give the percentage of implied solutions to all solutions and implied non-solutions to all non-solutions, respectively. Since $N = 1$, all non-solutions are single-block here.

Fig. 8. Percentage of implied solutions and non-solutions

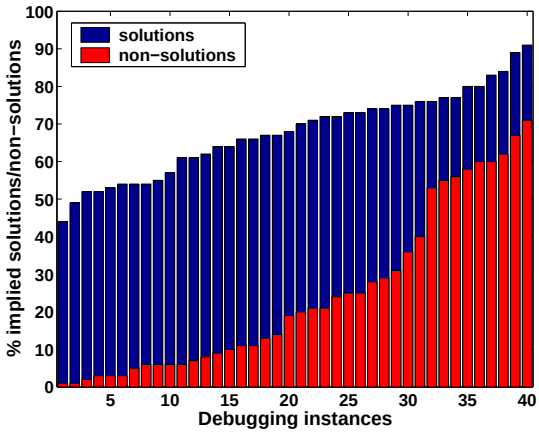


Figure 8 shows the percentages of implied solutions to all solutions and implied non-solutions to all non-solutions, sorted in increasing order. On average, 68% of all solutions are implied, resulting in a three-fold reduction in the number of SAT solver runs. This percentage of implied solutions goes up to 91% for vga-5, indicating that more than nine out of ten solutions are found without a SAT call. Figure 8 also shows that, on average, 25% of all non-solutions are implied by our tailored SAT solver. Figure 9 plots the number of found solutions versus run-time for **trad**, +**impl** and +**impl**−**impl** for design1-2. It can be seen that while **trad** returns solutions at roughly equal time intervals, +**impl** and +**impl**−**impl** discover solutions at a much faster rate due to solution implications. The rate of solution discovery decreases for both with time, mainly because implied solutions in later SAT calls might have already been found (or implied) in previous calls. Returning most solutions early is beneficial because the engineer can start examining returned solutions earlier, while the debugger continues to run. Moreover, as expected, Figure 9 demonstrates that non-solution implications speed-up SAT calls in +**impl**−**impl** compared to +**impl**.

Figure 10 plots the SAT run-times of each of +**impl**, −**impl** and +**impl**−**impl** versus those of **trad** on a logarithmic scale, along with the 1x, 2x, 3x and 10x lines, clearly showing the superiority of our approaches. The geometric means of the speed-ups from **trad** to each of +**impl**, −**impl** and +**impl**−**impl** are shown in the last line of Table II to be 2.07x, 1.45x and 2.63x excluding common overhead, and 1.75x, 1.33x and 2.02x including overhead. In most cases, higher percentages of implied solutions and non-solutions mean less and faster SAT calls, which result in less total SAT solving time. For instance, in design1-2, 80% of solutions and 58% of non-

TABLE II

RTL DEBUG USING DOMINANCE, $N = 1$

Instance	Alg. 1	+impl			−impl			+impl−impl				
	(s)	SAT (s)	impr (x)	impr-o (x)	SAT (s)	impr (x)	impr-o (x)	sol impl (%)	non-sol impl (%)	SAT (s)	impr (x)	impr-o (x)
vga-1	0.3	12.4	1.1x	1.1x	10.8	1.3x	1.2x	54%	21%	6.7	2x	1.6x
vga-2	1.1	80.7	2.6x	2.4x	202.4	1.1x	1.1x	77%	14%	75.0	2.8x	2.6x
vga-3	0.9	15.2	2.7x	2x	54.9	0.8x	0.8x	76%	6%	24.1	1.7x	1.5x
vga-4	2.2	24.5	2.5x	1.7x	54.7	1.2x	1.1x	90%	9%	35.0	1.8x	1.4x
vga-5	2.3	45.8	2.3x	1.6x	16.5	5.8x	2.2x	91%	36%	7.5	11.2x	2.5x
ddr2-1	0.2	46.9	2x	1.9x	70.8	1.4x	1.3x	54%	60%	40.6	2.4x	2.2x
ddr2-2	0.2	40.6	2.2x	2x	69.5	1.3x	1.3x	55%	56%	41.6	2.2x	2x
ddr2-3	0.2	31.6	2.8x	2.5x	63.4	1.4x	1.4x	72%	67%	36.1	2.4x	2.2x
ddr2-4	0.2	35.3	2.2x	2x	95.8	0.8x	0.8x	52%	21%	45.0	1.7x	1.6x
dma-1	2.0	170.4	2.5x	2.1x	323.5	1.3x	1.3x	77%	28%	161.9	2.7x	2.2x
dma-2	0.7	87.1	2.2x	2.1x	56.9	3.4x	2.9x	75%	40%	37.5	5.1x	3.9x
dma-3	0.7	64.8	1.7x	1.6x	87.8	1.2x	1.2x	66%	24%	45.3	2.4x	2.1x
dma-4	0.3	33.5	1.9x	1.7x	10.7	5.9x	3.4x	74%	62%	6.5	9.6x	4.1x
dma-5	2.1	38.3	1.8x	1.5x	48.3	1.4x	1.3x	53%	19%	31.3	2.2x	1.7x
dma-6	2.0	23.8	2.3x	1.5x	53.2	1.1x	1x	62%	29%	25.8	2.1x	1.5x
mips789-1	0.6	36.4	3x	2.5x	95.8	1.1x	1.1x	80%	1%	45.0	2.4x	2.1x
mips789-2	0.5	67.9	1.7x	1.6x	9.2	12.1x	4.6x	83%	55%	4.8	22.1x	5.4x
mips789-3	0.5	16.6	2.4x	1.9x	30.4	1.3x	1.2x	62%	6%	17.2	2.3x	1.8x
usb-1	0.2	39.5	2.4x	2.2x	56.9	1.7x	1.6x	57%	31%	37.5	2.5x	2.3x
usb-2	0.2	343.6	1.5x	1.5x	150.4	3.4x	3.1x	64%	60%	102.6	5x	4.3x
usb-3	0.2	8.2	2.4x	1.9x	15.9	1.3x	1.2x	61%	25%	8.6	2.3x	1.8x
usb-4	0.0	7.0	2.4x	1.7x	14.4	1.1x	1.1x	67%	11%	8.3	2x	1.5x
usb-5	0.2	8.2	2.4x	1.8x	16.0	1.2x	1.2x	54%	7%	10.4	1.9x	1.5x
usb-6	0.3	11.7	2.4x	1.9x	22.0	1.3x	1.2x	70%	53%	9.4	3x	2.2x
rsdecoder-1	0.2	28.9	2.1x	2x	87.8	0.7x	0.7x	44%	71%	45.3	1.3x	1.3x
rsdecoder-2	0.2	T/O	N/A	N/A	T/O	N/A	N/A	59%	14%	2940.5	∞	∞
hpdmc-1	0.0	4.7	2.3x	1.9x	6.5	1.7x	1.5x	64%	25%	3.8	2.9x	2.2x
hpdmc-2	0.0	4.7	1.8x	1.6x	6.7	1.3x	1.2x	66%	8%	3.7	2.3x	1.8x
hpdmc-3	0.0	7.7	2.2x	2x	9.3	1.9x	1.7x	73%	20%	5.7	3x	2.6x
hpdmc-4	0.0	3.7	2.3x	2x	5.3	1.6x	1.5x	72%	6%	2.8	3x	2.4x
hpdmc-5	0.0	11.2	1.4x	1.4x	11.4	1.4x	1.4x	49%	13%	7.3	2.2x	2x
hpdmc-6	0.0	2.6	2.4x	2x	4.4	1.4x	1.3x	67%	11%	2.6	2.4x	2x
mrisc-1	0.1	5.6	2x	1.5x	9.6	1.2x	1.1x	76%	3%	4.5	2.5x	1.7x
mrisc-2	0.1	7.0	2.3x	1.7x	14.5	1.1x	1.1x	73%	2%	6.6	2.4x	1.7x
mrisc-3	0.1	5.9	2.4x	1.7x	13.4	1.1x	1.1x	74%	3%	5.7	2.5x	1.8x
design1-1	2.8	1351.8	1.4x	1.4x	1621.1	1.2x	1.2x	61%	6%	1095.2	1.8x	1.7x
design1-2	0.0	45.7	2.5x	2.3x	63.9	1.8x	1.7x	80%	58%	27.4	4.1x	3.6x
design1-3	0.4	34.4	3.3x	2.3x	100.2	1.1x	1.1x	84%	3%	35.9	3.2x	2.2x
design2-1	3.1	6.4	1.2x	1.1x	10.5	0.9x	0.9x	68%	10%	6.4	1.2x	1.1x
design2-2	3.1	2.7	1x	1x	6.0	0.7x	0.9x	52%	1%	3.0	1x	1x
design2-3	3.1	12.2	1.3x	1.1x	16.2	1x	1x	75%	5%	10.3	1.5x	1.2x
mean			**2.07x**	**1.75x**		**1.45x**	**1.33x**	**68%**	**25%**		**2.63x**	**2.02x**

TABLE III

RTL DEBUG USING DOMINANCE, $N = 3$

Instance	$\lvert\mathbf{n}\rvert$	$\lvert\mathcal{B}\rvert$	k	# sol	overh (s)	trad SAT (s)	Alg. 1 (s)	+impl−impl($n=1$) SAT (s)	impr (x)	impr-o (x)	+impl−impl($n=1,2,3$) SAT (s)	impr (x)	impr-o (x)
vga-6	90759	1850	172	222	13.2	11.4	0.2	4.7	2.3x	1.4x	4.4	2.4x	1.4x
vga-7	89626	1534	34	166	7.4	111.0	2.3	48.2	2.2x	2x	48.3	2.2x	2x
ddr2-5	58399	2779	29	373	19.0	78.4	0.2	37.4	2.1x	1.7x	37.5	2.1x	1.7x
usb-7	36180	3477	18	165	19.1	20.3	0.2	7.6	2.6x	1.5x	7.5	2.6x	1.5x
hpdmc-7	11879	1079	48	219	6.6	11.8	0.0	6.0	1.9x	1.4x	4.5	2.6x	1.6x
hpdmc-8	11061	961	45	207	10.6	8.4	0.0	4.1	2.1x	1.3x	3.5	2.4x	1.3x
hpdmc-9	11548	1080	48	192	2.0	17.9	0.0	8.6	2.1x	1.9x	8.5	2.1x	1.9x
hpdmc-10	12849	1135	25	152	13.9	8.5	0.0	2.8	3x	1.3x	2.8	3x	1.3x
mrisc-4	18052	968	41	182	12.7	16.1	0.1	6.7	2.4x	1.5x	6.6	2.4x	1.5x
mrisc-5	17526	958	40	163	7.0	14.6	0.1	6.0	2.4x	1.6x	5.7	2.5x	1.7x
design2-4	875837	2863	58	1491	43.7	532.0	3.1	337.5	1.6x	1.5x	308.2	1.7x	1.6x
mean									**2.21x**	**1.55x**		**2.35x**	**1.59x**

solutions are implied, yielding a 4.1x speed-up in total SAT run-time, compared to the average 2.63x speed-up. However, this is not always true because of the unpredictable behavior of SAT solvers. Furthermore, we have not found any clear relationships between design parameters and improvements due to solution and non-solution implications.

Finally, Table III shows experimental results for eleven debugging instances with $N = 3$. Here, **trad** is compared to two versions of +**impl**−**impl**, first only allowing non-solutions of $n = 1$ blocks, then non-solutions of up to $n = 3$

blocks. The columns of Table III are structured similarly to those of Tables I and II. We can see that detecting and implying non-solutions of up to three blocks increases the geometric mean of the speed-up of +**impl**−**impl** relative to **trad** from 2.28x to 2.42x excluding common overhead, and from 1.55x to 1.58x including overhead.

VIII. Conclusion

We first presented an iterative algorithm for computing dominance relationships between the blocks of an RTL design. We showed how to leverage these dominance relationships in an automated RTL debugger to expedite the discovery of potentially buggy RTL blocks, as well as the clearance of blocks guaranteed to be correct. This was done by using solution and non-solution implications. Our methods reduced the number of SAT calls three-fold and speed-up each call, resulting in a 2.63x overall speed-up in total SAT solving time, as demonstrated on an extensive set of experiments on industrial designs.

In the future, we plan to use dominance relationships between RTL blocks to group and rank all potential bugs.

References

[1] H. Foster, "From volume to velocity: the transforming landscape in function verification," in *Design Verification Conference*, 2011.

[2] *International technology roadmap for semiconductors*, 2011. [Online]. Available: http://www.itrs.net/Links/2011ITRS/2011Chapters/2011Design.pdf

[3] D. McGrath, "De Geus touts new products, says ICs will rebound," in *EE Times*, March 2009.

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

[5] T. Lee, W. Chuang, I. Hajj, and W. Fuchs, "Circuit-level dictionaries of CMOS bridging faults," in *IEEE VLSI Test Symposium*, 1994, pp. 386–391.

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

[7] S.-Y. Huang and K.-T. Cheng, "Errortracer: design error diagnosis based on fault simulation techniques," *IEEE Transactions on CAD*, vol. 18, no. 9, pp. 1341–1352, 1999.

[8] J. Liu and A. Veneris, "Incremental fault diagnosis," *IEEE Transactions on CAD*, vol. 24, no. 2, pp. 240–251, 2005.

[9] A. Smith, A. Veneris, and A. Viglas, "Design diagnosis using Boolean satisfiability," in *Asia and South Pacific Design Automation Conference*, 2004, pp. 218–223.

[10] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M. Abadir, "Debugging sequential circuits using Boolean satisfiability," in *International Conference on CAD*, 2004, pp. 204–209.

Fig. 9. # solutions vs. run-time for design1-2

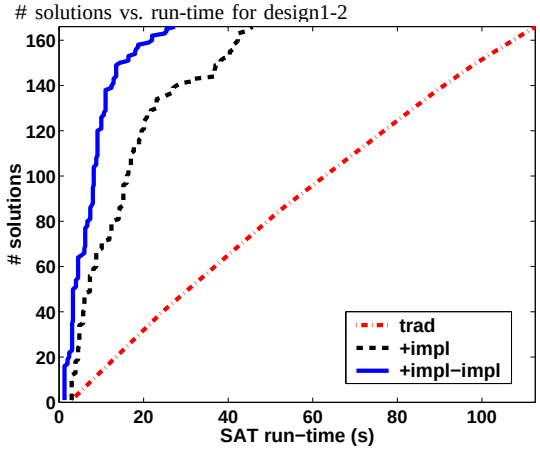
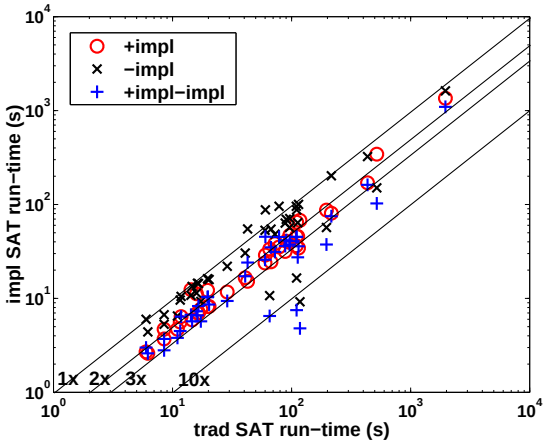


Fig. 10. Performance Results



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

[12] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *International Conference on CAD*, 2005, pp. 871–876.

[13] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug, and test," *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 981–994, 2010.

[14] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Transactions on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.

[15] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Transactions on CAD*, vol. 29, no. 11, pp. 1790–1803, 2010.

[16] B. Keng and A. Veneris, "Scaling VLSI design debugging with interpolation," in *International Conference on Formal Methods in CAD*, 2009, pp. 144–151.

[17] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Transactions on CAD*, vol. 29, pp. 1804–1817, 2010.

[18] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2117–2132, 1999.

[19] L. Georgiadis and R. E. Tarjan, "Finding dominators revisited: extended abstract," in *ACM-SIAM Symposium on Discrete Algorithms*, 2004, pp. 869–878.

[20] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Design Automation Conference*, 1987, pp. 502–508.

[21] T. Niermann and J. H. Patel, "HITEC: a test generation package for sequential circuits," in *European Design Automation Conference*, 1991, pp. 214–218.

[22] R. Gupta, "Generalized dominators and post-dominators," in *Symposium on Principles of Programming Languages*, 1992, pp. 246–257.

[23] S. Alstrup, J. Clausen, and K. Jørgensen, "An $O(|V|*|E|)$ algorithm for finding immediate multiple-vertex dominators," *Information Processing Letters*, vol. 59, no. 1, pp. 9–11, 1996.

[24] M. Ganai, "Propelling sat and sat-based bmc using careset," in *International Conference on Formal Methods in CAD*. IEEE, 2010, pp. 231–238.

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

[26] M. Ganai and A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," in *Asia and South Pacific Design Automation Conference*, 2007, pp. 310–315.

[27] K. Cooper, T. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, pp. 1–10, 2001.

[28] M. Benedetti, A. Lallouet, and J. Vautard, "QCSP made practical by virtue of restricted quantification," in *International Joint Conference on Artificial Intelligence*, 2007, pp. 38–43.

[29] R. Krenz and E. Dubrova, "A fast algorithm for finding common multiple-vertex dominators in circuit graphs," in *Asia and South Pacific Design Automation Conference*, 2005, pp. 529–532.

[30] F. E. Allen and J. Cocke, "Graph-theoretic constructs for program flow

analysis," Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, Tech. Rep., 1972.
[31] J. Kam and J. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM*, vol. 23, no. 1, pp. 158–171, 1976.
[32] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
[33] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Design Automation Conference*, 2001, pp. 530–535.
[34] OpenCores.org, "http://www.opencores.org," 2007.

Hratch Mangassarian (S'01-M'12) received the B.E. degree in Computer and Communications Engineering with high distinction from the American University of Beirut in 2005, and the M.A.Sc. and Ph.D. degrees in Electrical and Computer Engineering from the University of Toronto in 2008 and 2012. His research is on formal verification and automated design debugging of digital designs, as well as QBF and its applications to CAD. He is a member of IEEE and ACM. He is currently with Google Inc.

Bao Le (S'13) received the B.A.Sc. degree in computer engineering from the University of Toronto in 2010, and the M.A.Sc. degree in computer engineering from the University of Toronto in 2012. 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 formal verification and automated design debugging of digital systems as well as evaluation and verification techniques for low-power designs.

Andreas Veneris (S'96-M'99-SM'05) 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 UrbanaChampaign 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 a 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 several patents. He is a member of IEEE, ACM, AMC, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.