Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment

Bao Le[1], Brian Keng[1], Hratch Mangassarian[1], Andreas Veneris[1,2]

***Abstract*—With the growing complexity of VLSI designs, functional debugging has become a bottleneck in modern CAD flows. To alleviate this cost, various SAT-based techniques have been developed to automate bug localization in the RTL. In this context, dominance relationships between circuit blocks have been recently shown to reduce the number of SAT solver calls, using the concept of *solution implications*. This paper first introduces the dual concepts of *reverse domination* and *non-solution implications*. A SAT solver is tailored to leverage reverse dominators for the early on-the-fly detection of bug-free components. These are non-solution areas and their early pruning significantly reduces the the debugging search-space. This process is expedited by branching on *error-select* variables first. Extensive experiments on tough real-life industrial debugging cases show an average speedup of 1.7x in SAT solving time over the state-of-the-art, a testimony of the practicality and effectiveness of the proposed approach.**

I. Introduction

Design errors are becoming increasingly common with the growing complexity of VLSI designs. Design debugging is the process of localizing the bug(s) in the RTL, based on a failing counter-example trace. Today, bigger designs and longer traces have made debugging a resource-intensive task, which consumes up to 60% of the total verification effort [1].

As a result, various methodologies have been proposed to automate design debugging and reduce its cost [2]–[7]. Due to advancements in formal engines, most modern debugging techniques use Boolean Satisfiability (SAT) solvers [5]. The problem is encoded as a SAT instance, where each satisfying assignment corresponds to a potential bug location, called a *solution* [8]. Each solution consists of a (set of) circuit block(s) or RTL line(s), that can be modified to fix the erroneous behavior in the counter-example trace. All-solution SAT-based debugging guarantees that the root cause of the error is one of these solutions, which greatly simplifies the task of identifying and fixing the actual bug.

With typical design sizes exceeding the half-million synthesized gates mark, the propositional formulas encoding design debugging can have tens of millions of variables and clauses [7]. This underlying complexity often presents a challenge even to modern SAT solvers. The motivation behind this work is to prune the search-space of the all-solution SAT solver in design debugging. This is done by leveraging *dominance relationships* between circuit blocks. A block a is said to dominate another block b if every path from every node in b to a primary output passes through a node in a. Dominators have been used to optimize various CAD tasks, *e.g.*, test pattern generation and verification [9]–[11]. Recently, dominance between circuit blocks has been successfully used in an automated RTL debugger [12] to reduce the number of SAT solver calls by introducing the concept of *solution implications*.

This work makes use of the concept of reverse domination. In more detail, we say that block b is a *reverse dominator* of block a if a dominates b. It is shown that if a is not part of any solution, then all its reverse dominators can also be ruled out as *non-solutions*, that is, as blocks that cannot be modified in any way to correct the counter-example trace. Based on this idea, we tailor a SAT solver to leverage reverse dominators for performing *non-solution implications*. We present a new *SAT branching scheme*, where *error-select* [5], [13] variables are decided upon first. This allows us to learn blocks that are not part of any solution early in the solving process. Hence, the concept of reverse domination could be used much more effectively.

The presented techniques are implemented in a SAT-based automated RTL debugger, using MiniSat 2.2.0 as the back-end solver. An extensive set of experiments on real industrial designs demonstrates that performing both solution and non-solution implications results in an average speedup of 1.7x in SAT solving time over performing only solution implications [12]. These results demonstrate the effectiveness and practicality of our contributions.

The paper is organized as follows. Section II contains background on design debugging and block dominance. Section III presents the theory for leveraging reverse block dominators to perform non-solution implications. Section IV gives our SAT branching algorithm, which makes non-solution implications effective. Section V shows experimental results and Section VI concludes the paper.

II. Preliminaries

The following notation is used throughout the paper. Given a sequential circuit C, the symbols x, y, and s respectively represent the sets of primary inputs, primary outputs and state elements (flip flops) in C. Let l denote the set of all nodes (including nodes in x, y, s). For each $z \in \{l, x, y, s\}$, the Boolean variable z_i denotes the ith element of set z.

For simplicity, we consider designs with a single clock-domain, but the theory developed here is applicable to multiple clock-domain designs using the results of [14]. *Time-frame expansion* is a modeling technique for sequential circuits, which replicates (*i.e.,* unrolls) the combinational components of C k times, such that the next state of each time-frame is connected to the current state of the next time-frame. For any variable z_i (or set z), z_i^t (or z^t) denotes the corresponding variable (or set) in time-frame t. The behavior of C during the tth clock-cycle is dictated by the transition relation predicate $T(s^t, s^{t+1}, x^t, y^t)$, which can be extracted from C and encoded in CNF

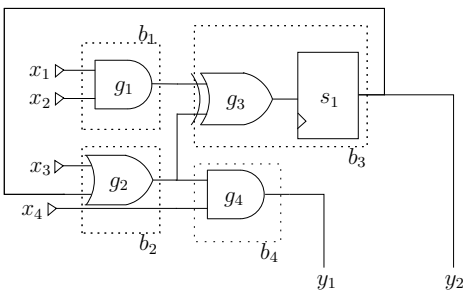

Fig. 1. A Sequential Circuit

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({lebao, briank, hratch, veneris}@eecg.toronto.edu)

[2]University of Toronto, CS Department, Toronto, ON M5S 3G4

978-3-9810801-8-6/DATE12/2012 EDAA

using Tseitin transformation with the auxiliary variables in l^t (*i.e.,* the logic gates) [15].

Some of the nodes in l are grouped into blocks. Each block consists of the synthesized gates corresponding to an RTL "block", such as an *if* statement or an *always* block. Let $B = \{b_1, b_2, ..., b_{|B|}\}$ denote the set of all blocks, where each $b_i \subseteq l$. Note that the same node l_i could belong to more than one block because of the hierarchical nature of RTL. The set $out(b_i)$ includes the outputs of block b_i. In the unrolled circuit, the set b_i^t denotes set of nodes belonging to block b_i in time-frame t. Consequently, $out(b_i^t)$ is the set of outputs of block b_i at time-frame t.

A. Design Debugging

This section describes SAT-based design debugging and introduces relevant notation. Given an erroneous design, a counter-example of length k and an error cardinality N, the goal of an automated design debugger is to find all sets of N blocks that can potentially be responsible for the faulty behavior associated with the counter-example. Each such set is referred as a $solution$ of cardinality N. SAT-based design debugging [5], [13] encodes the problem as a propositional formula, where each satisfying assignment corresponds to a solution. The encoding process consists of the following steps.

First, a set of *error-select* variables $e = \{e_1, \ldots, e_{|B|}\}$ is added to the circuit, where each e_i is associated with a block b_i. The circuit is modified such that setting $e_i = 1$ disconnects the nodes in $out(b_i)$ from their fanins, making them free variables, while setting $e_i = 0$ does not modify the circuit. Next, time-frame expansion is performed on this enhanced circuit, such that $out(b_i^t)$ are controlled by the same error-select variable e_i, for all time-frames t. This allows the SAT solver to modify the outputs of block b_i across all time-frames by setting $e_i = 1$ to "fix" any potential errors in b_i.

Then, constraints are applied to the initial state, primary inputs and primary outputs. These constraints ensure that given the initial state $\Phi_S(s^1)$ and primary inputs $\Phi_X(x^1, ..., x^k)$ from the counter-example, the enhanced circuit produces the *expected* outputs $\Phi_Y(y^1, ..., y^k)$. Finally, an error cardinality constraint $\Phi_N(e)$ is added to enforce $\Sigma_{i=1}^{|B|} e_i = N$. Overall, the design debugging problem is encoded as:

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

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

Each assignment to $e = \{e_1, \ldots, e_{|B|}\}$ satisfying $Debug$ (1) corresponds to a debugging solution, and the SAT solver must find *all* such satisfying assignments to e. This is normally done by iteratively blocking each satisfying assignment using a blocking clause and re-solving $Debug$ until the problem becomes unsatisfiable or UNSAT.

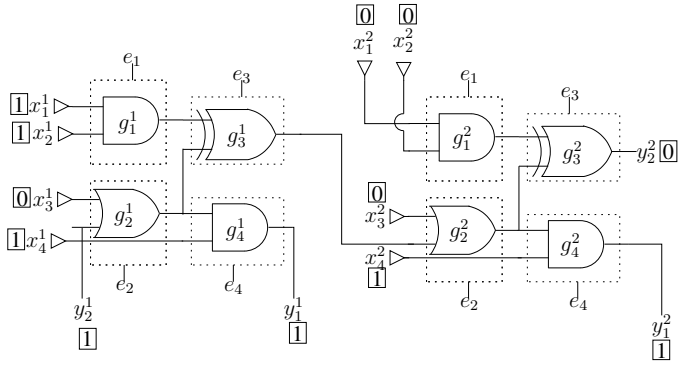

Fig. 2. Design Debugging Formulation

Example 1 *Consider the sequential circuit presented in Figure 1. We are also given a two-cycle counter-example with initial state $s_1 = 1$, inputs $\langle x_1, x_2, x_3, x_4 \rangle = \langle \langle 1, 1, 0, 1 \rangle, \langle 0, 0, 0, 1 \rangle \rangle$ and expected outputs $\langle y_1, y_2 \rangle = \langle \langle 1, 1 \rangle, \langle 1, 1 \rangle \rangle$, demonstrating a mismatch in the second time-frame at the output y_1.*

The corresponding design debugging formulation is illustrated in Figure 2. As shown, each block b_i is associated with an error-select variable e_i. The initial-state/input/output constraints are shown in boxes. The constraint Φ_N is omitted for brevity. For $N = 1, \{b_4\}$ is returned by the automated design debugger as the only solution. b_4 is indeed the buggy block and could be corrected by turning gate g_4 into an OR *gate.*

B. Block Dominance

Block b_j is said to dominate block b_i if every path from a node in $out(b_i)$ to a primary output contains a node in b_j. The notation $b_j \mathsf{D} b_i$ indicates that b_j dominates b_i, where D is referred to as the block *dominance relation*. Furthermore, the set $\mathsf{D}(b_i) = \{b_j | b_j \mathsf{D} b_i\}$ consists of blocks that dominate b_i.

Example 2 *Consider the sequential circuit in Figure 1. Block b_3 dominates block b_1 while no other blocks dominate any other blocks. This is because every path from $out(b_1)$ has to pass through gate g_3 of b_3 to reach the primary outputs y_1, y_2. However, block b_4 does not dominate any blocks because there exist paths from b_1, b_2 and b_3 to primary output y_2 that do not pass through b_4.*

The authors of [12] discuss why existing methods for computing so-called single and multiple-vertex dominators are not applicable in a design debugging setting, and present a fixpoint algorithm for computing the block dominance relation D. The run-time of their algorithm is $O(c \cdot |B| \cdot |E|)$, where $|B|$ is the number of blocks, $|E|$ is the number of edges in C and c is called the loop-connectedness of C.

Furthermore, [12] proves that given a solution $\{b_{i_1}, ..., b_{i_N}\}$ of $Debug$ (1), if $\bigwedge_{n=1}^{N} (b_{j_n} \mathsf{D} b_{i_n})$, then $\{b_{j_1}, ..., b_{j_N}\}$ is also a solution. This allows them to leverage the block dominance relation D to perform *solution implications*, which significantly reduces the number of SAT calls and speeds up the debugging process.

III. Non-Solution Implications using Reverse Domination

In this section, we first define reverse dominators and non-solution blocks. Next, we prove that reverse dominators can be leveraged to perform non-solution implications, given an original non-solution block. In the following section, we present a branching heuristic which enables the SAT solver to find original non-solutions much faster, leading to earlier non-solution implications.

Definition 1 *A block b_i is a reverse dominator of block b_j, denoted as $b_i \mathsf{D}^{-1} b_j$, if and only if $b_j \mathsf{D} b_i$.*

Clearly, the reverse block dominance relation D^{-1} is completely determined by D, which can be computed using the algorithm in [12]. The set $\mathsf{D}^{-1}(b_j) = \{b_i | b_i \mathsf{D}^{-1} b_j\}$ consists of reverse dominators of b_j, *i.e.,* the blocks that b_j dominates.

Definition 2 *Given an erroneous design C, a counter-example of length k along with the corresponding expected outputs and an error cardinality N, b_i is a non-solution block if and only if $Debug \wedge e_i$ is* UNSAT.

In other terms, a non-solution block cannot be part of any solution of cardinality N. If $N = 1$, then a non-solution block cannot be modified

in any way to correct the erroneous behavior in the counter-example trace. We will prove that reverse dominators of non-solution blocks are also non-solution blocks. In order to do so, we need to prove the following lemma.

Lemma 1 *Given an erroneous design C, a counter-example of length k along with the corresponding expected outputs and an error cardinality N, we have:*

$$((Debug \wedge e_i \textit{ is } \texttt{SAT}) \wedge b_j \mathsf{D} b_i) \Rightarrow (Debug \wedge e_j \textit{ is } \texttt{SAT})$$

Proof: Let π denote the satisfying assignment of $(Debug \wedge e_i)$. Assuming that $b_j \mathsf{D} b_i$, we will construct an assignment π' satisfying $(Debug \wedge e_j)$.

We first construct $\pi'(e)$. Let the set of error-select variables assigned to 1 in $\pi(e)$ be $\{e_i, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\}$, where $\{\sigma_1, \ldots, \sigma_{N-1}\} \subseteq [1, |B|] - \{i\}$.

1) If $j \notin \{\sigma_1, \ldots, \sigma_{N-1}\}$, we let the set of error-select variables assigned to 1 in $\pi'(e)$ be $\{e_j, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\}$.
2) If $j \in \{\sigma_1, \ldots, \sigma_{N-1}\}$, we let the set of error-select variables assigned to 1 in $\pi'(e)$ be $\{e_i, e_{\sigma_1}, \ldots, e_{\sigma_{N-1}}\}$.

In both cases, the number of error-select variables assigned to 1 in $\pi'(e)$ is N, satisfying Φ_N.

Since $b_j \mathsf{D} b_i$, any path from $out(b_i)$ to a primary output must pass through $out(b_j)$. This makes it possible to partition the unrolled enhanced circuit described in Subsection II-A into two parts: Let I refer to the sub-circuit in the fan-out cone of $out(b_i)$ (that fans out to $out(b_j)$) and let J refer to the rest of the circuit (excluding error-select variables). In $Debug \wedge e_j$, clearly $\pi'(e_j) = 1$ in both cases shown above, effectively disconnecting $out(b_j)$ from its fanins. As such, $out(b_i)$ is disconnected from the primary outputs. A node is said dangling if there is no path from such node to the primary outputs. Hence, $out(b_i)$ and becomes dangling logic. This means that I is dangling (although J can fan-out to I). Since there are no external constraints on I, $\pi'(I)$ can be computed by simply "propagating" whatever $\pi'(out(b_i))$ and $\pi'(J)$ are into I (using gate propagation, which is effectively unit propagation in CNF). Hence, what remains is to construct $\pi'(J)$.

Note that every error-select variable e_k other than e_i or e_j is assigned to the same value in π and π', as shown in both cases above. Furthermore, since $\pi'(e_j) = 1$, we are free to set $\pi'(out(b_j)) = \pi(out(b_j))$. In addition, recall that $out(b_i)$ has no effect on J since I is dangling. As such, since $\pi'(e_k) = \pi(e_k)$ for all other e_k, for all nodes $v \in out(b_k) \cap J$, we can simply set $\pi'(v) = \pi(v)$. As a result, $\pi'(J) = \pi(J)$. Since $\pi(J)$ satisfies all the constraints in $Debug$, so does $\pi'(J)$. Finally, since $\pi'(e_j) = 1$, π' satisfies $Debug \wedge e_j$. ■

The following theorem proves that reverse dominators can be used to perform non-solution implications.

Theorem 1 *Given an erroneous design C, a counter-example of length k along with the corresponding expected outputs and an error cardinality N, if b_j is a non-solution block of Debug and $b_i \mathsf{D}^{-1} b_j$, then b_i is also a non-solution block of Debug.*

Proof: To clarify the presentation, let us define the predicates Φ_i and Φ_j, as follows:

$$\Phi_i = Debug \wedge e_i \text{ is } \texttt{SAT} \qquad \Phi_j = Debug \wedge e_j \text{ is } \texttt{SAT}$$

Using Lemma 1, we have:

$$\begin{aligned} & (\Phi_i \wedge b_j \mathsf{D} b_i) \Rightarrow \Phi_j \\ \Leftrightarrow \quad & \neg\Phi_i \vee \neg (b_j \mathsf{D} b_i) \vee \Phi_j \\ \Leftrightarrow \quad & \neg\Phi_i \Leftarrow (b_j \mathsf{D} b_i \wedge \neg\Phi_j) \\ \Leftrightarrow \quad & \left(b_i \mathsf{D}^{-1} b_j \wedge \neg\Phi_j\right) \Rightarrow \neg\Phi_i \end{aligned}$$

■

Example 3 *Consider the sequential circuit in Figure 1 and the debugging problem presented in Example 1. We know that block b_3 is a dominator of block b_1 from Example 2. If b_3 is known to be a non-solution, using Theorem 1, we know that b_1 is also a non-solution. We can therefore automatically add the clause $(\neg e_1)$ to prune the search-space of Debug.*

Next, we explain how to make use of Theorem 1. An all-solution SAT solver returns when a solution is found, making it possible to imply its dominating solutions [12] without modifying the SAT solver, add them to $Debug$ using blocking clauses and continue solving. However, the SAT solver usually does not learn non-solution blocks until the end of the solving process. This hinders the application of Theorem 1. As such, it is necessary to tailor the SAT solver to recognize when a non-solution block has been *learned*.

Watching for learned clauses of the form $(\neg e_i)$ is not desirable because the SAT solver rarely learns such unit clauses. Instead, learned clauses are much more complex and usually involve many other variables along with error-select variables. Another way to realize that a block is a non-solution is to examine the forced assignments of unit propagation (BCP) after each solver restart (when the decision stack is empty). If some $e_i = 0$ by unit propagation given an empty decision stack, then the solver has "learned" that b_i is a non-solution block. However, from our experience, using a generic SAT solver, virtually all 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.

The following section shows how to modify the branching scheme of the SAT solver to overcome this problem.

IV. SAT Branching Scheme for Early Non-Solution Learning

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 using reverse dominators useful.

A. SAT Branching Scheme

The decision tree in a SAT solver gives the order in which variables are decided upon. 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 $e_i = 1$, the internal nodes of block b_i become dangling, and therefore they are don't-cares. As such, assigning the nodes in b_i, as well as their fanouts, is useless if e_i is later assigned to 1.

A 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 due to reverse dominance to prune the SAT search-space earlier and therefore more effectively. Subsection IV-B discusses how to detect learned non-solutions using our branching scheme.

As a result, we force the SAT solver to first decide on all error-select variables (e). Furthermore, we force the solver to always assign error-select variables that are decided (*i.e.,* not forced due to Φ_N) to 1 before trying to set them to 0. The reason for doing this is to learn non-solutions, and is explained in detail in Subsection IV-B. Once all the error-select variables are assigned, the solver uses the standard decision heuristics (*e.g.,* VSIDS [16]) for the remaining variables.

In addition, modern SAT solvers have periodic *restarts*, usually after a certain number of conflicts. We take advantage of this 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 move all the error-select variables above level r in the decision tree under those that are below level r. The reasoning behind this is to avoid being engaged in parts of the search-space where it is hard to learn new non-solutions.

Algorithm 1: SAT Solver for Design Debugging

input: CNF $Debug$, Dominator relation D, set e

```
1  foreach e_i ∈ e do Priority(e_i) ← ∞;
2  learned ← true;
3  result ← BCP ();
4  while result ≠ (SAT/UNSAT) do
5      heap ← buildHeap (Priority) ;
6      if learned = false then
7          r ← genRandom (1, |B|);
8          heap ← reorderHeap (r);
9      end
10     learned ← false;
11     numConf ← 0;
12     e_i ← heap.firstErrorSelect ();
13     while numConf < maxConf do
14         if result = (SAT/UNSAT) then return ;
15         if result = Conflict then
16             numConf ++;
17             resolveConflict ();
18         end
19         next ← heap.pop ();
20         if next ∈ e then next.assign (1);
21         else next.assign (polarity ());
22         if (e_i.value() = 0) then
23             // b_i is the block e_i represents
24             foreach e_j ∈ D(b_i) do
25                 Debug ← Debug ∧ (¬e_j);
26             end
27             e_i ← nextErrorSelect ();
28             learned ← true;
29         end
30         result ← BCP ();
31     end
32 end
```

B. Detecting Learned Non-Solution Blocks

To simplify the presentation of this subsection, let us assume without loss of generality that the variable at the root of the decision tree is e_1. According to our branching scheme explained in the previous section, the SAT solver first assigns $e_1 = 1$. If the solver later switches to $e_1 = 0$ without finding a satisfying assignment under $e_1 = 1$, this means that $e_1 = 1$ cannot be extended to a satisfying assignment. Hence, $e_1 = 0$ is true for all satisfying assignments (if any exist). In other terms, $(\neg e_1)$ has been learned and b_1 is a non-solution block.

This observation is not applicable to all non-root variables in the decision tree. Consider variable e_2 in the subtree under $e_1 = 1$, switching from $e_2 = 1$ to $e_2 = 0$ without finding a satisfying assignment does not imply that $(\neg e_2)$ has been learned. However, it is possible to learn about non-root variables in some circumstances, as shown by Lemma 2.

Lemma 2 *Using the branching scheme given in Subsection IV-A, until a satisfying assignment is found, all the error-select variables set to* 0 along the right-most path of the decision tree *correspond to non-solution blocks.*

Proof: Assume that the error-select variables are decided in the order of $\langle e_1, \ldots, e_{|B|} \rangle$. Recall that our branching scheme forces the solver to first set each error-select variable to 1 before trying to set it to 0. Also assume that $e_1 = 0, \ldots, e_j = 0$ have been set along the right-most path of the decision tree and no satisfying assignment has been found yet. Then by construction, all other assignments to $e_1, \ldots, e_j$ have been examined and setting any of them to 1 cannot be extended to a satisfying assignment. In other terms, each of $Debug \wedge e_1, \ldots, Debug \wedge e_j$ is UNSAT. By Definition 2, this means that each of $b_1, \ldots, b_j$ is a non-solution block. ■

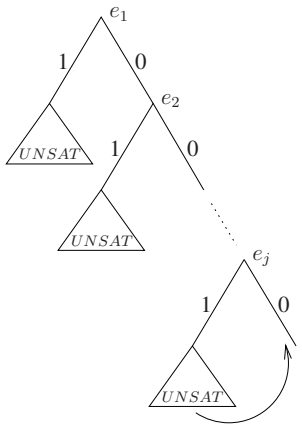

Fig. 3. Non-solution blocks using our branching scheme

Note that forced variables (due to BCP) are not part of the decision tree. Using Lemma 2, as soon as the SAT solver switches from $e_j = 1$ to $e_j = 0$, as long as all its ancestors in the decision tree are assigned to 0 and no satisfying assignment has been found yet, we can be sure that b_j is a non-solution block. This scenario is shown in Figure 3. Using this, we can imply that every block $b_i \in \mathsf{D}^{-1}(b_j)$ is also a non-solution, by Theorem 1, and therefore add the clause $(\neg e_i)$ for each reverse dominator.

C. Overall Modified SAT Algorithm

Algorithm 1 presents the pseudocode of our modified SAT solver. All unassigned variables are already assumed to have been assigned *priority* values, which set their order in the decision tree. Our algorithm assigns error-select variables very large priority values on line 1, in order to guarantee that they will be at the top of the decision maxheap [17] built on line 5, which is used to pick the next decision variable.

On line 12, the unassigned error-select variable with the highest priority is stored in e_i. The next variable is popped from the heap on line 19. If this variable $next$ is an error-select line, then it must be first assigned to 1 (line 20), otherwise the function $polarity()$ decides the polarity of $next$ using heuristics such as VSIDS [16] (line 21). Later, if e_i is assigned to 0, block b_i is learned as a non-solution block. As a result, each b_j that is dominated by b_i is also learned as a non-solution block and the unit clause $(\neg b_j)$ is added (line 25). After b_i is learned as a non-solution, e_i is updated so that new non-solutions can be learned (line 27). Other functions of the SAT engine such as $BCP()$ and $resolveConflict()$ are not modified.

As mentioned in Subsection IV-A, in order to direct the solver towards parts of the search-space where it is easier to learn non-solutions, we use the variable $learned$. If $learned = false$ on line 6, the SAT engine has spent a full iteration under $e_i = 1$ without learning any new non-solutions (or finding a solution). In this case, the heuristic is applied to reorder the heap such that the SAT engine would not pick e_i first in the next restart. A random number r ($1 \leq r \leq |B|$) is generated, and all the error-select variables above level r in the decision tree move below the ones under level r.

V. Experimental Results

This section presents the experimental results for the proposed framework on industrial design debugging problems. All experiments

TABLE I
DESIGN DEBUGGING SAT SOLVER RESULTS

Instance Info					dbg-dom	dbg-dom+rev			dbg-dom+rev+RR		
instance	k	$\|l\|$	$\|B\|$	# sols	time (s)	time (s)	# impl non-sols	imprv (x)	time (s)	# impl non-sols	imprv (x)
rsdecoder1	112	13543	2044	430	T/O	6955.90	1192	∞	5502.49	1211	∞
rsdecoder2	112	13564	2044	396	33.35	20.46	941	1.6x	19.11	798	**1.7x**
usb_funct1	32	35158	3425	422	53.17	45.46	631	**1.2x**	48.15	941	1.1x
usb_funct2	53	35350	4201	576	134.46	117.83	1167	1.1x	110.65	1487	**1.2x**
wb_dma1	35	191386	7896	468	123.89	97.26	2100	1.3x	81.34	1823	**1.5x**
wb_dma2	7	299838	8460	205	49.14	36.90	3384	1.3x	34.92	2357	**1.4x**
wb_dma3	28	299862	8836	526	304.18	182.09	5135	1.7x	179.51	3241	**1.7x**
vga1	423	89412	1593	128	434.81	172.51	145	**2.5x**	200.75	192	2.2x
vga2	423	89402	1741	84	106.98	147.95	277	0.7x	66.33	135	**1.6x**
ucrc_par	155	1056	63	20	7.97	3.94	0	**2.0x**	6.38	0	1.3x
mem_ctrl1	581	48006	3355	23	12.53	24.67	567	0.5x	7.13	63	**1.8x**
mem_ctrl2	1180	48006	3355	9	11.76	4.78	0	2.5x	4.76	0	**2.5x**
mips7891	153	30711	953	49	22.08	13.51	53	**1.6x**	15.80	34	1.5x
opensparc_ddr21	29	58399	2792	373	48.45	33.42	1072	1.4x	29.81	1190	**1.6x**
opensparc_ddr22	27	64915	2791	509	44.11	39.39	1138	1.1x	36.57	1394	**1.2x**
design1-1	71	499325	20204	69	53.40	25.08	40	2.1x	20.15	556	**2.7x**
design1-2	26329	499705	20211	117	72.54	38.27	5073	1.9x	38.62	5886	**1.9x**
design1-3	5343	499696	20209	120	39.63	31.69	210	1.3x	29.52	199	**1.4x**
design1-4	467	499705	20211	150	100.89	45.69	5854	2.2x	42.42	5882	**2.4x**
design1-5	177	499705	20211	98	73.72	27.04	5760	**2.7x**	29.28	5665	2.6x
design2-1	26	45632	5507	61	18.47	14.59	543	1.3x	9.89	17	**1.9x**
design2-2	5	203706	7416	50	7.38	4.23	53	**1.7x**	4.76	59	1.6x
design2-3	20	2082	185	62	0.13	0.08	65	**1.6x**	0.09	72	1.4x
design3-1	56	5454	495	129	3.03	2.07	187	1.6x	1.94	153	**1.6x**
design3-2	144	2333	144	28	0.083	0.07	52	1.2x	0.066	19	**1.3x**
AVERAGE					73.17	47.03			42.49		

are run using a single core of a i5-2400 3.1 GHz workstation with 8GB of RAM and a timeout of 7200 seconds. The presented techniques are implemented on top of a state-of-the-art SAT-based debugger [5], [12], [13] with a Verilog front-end to allow for RTL diagnosis. We tailor the debugger's back-end solver, MINISAT 2.2.0 [18], to leverage reverse dominators for performing non-solution implications as described in this work.

Eight industrial Verilog designs from OpenCores [19] and three commercial designs provided by our industrial partners are used in our experiments. For each design, several debugging instances are generated by injecting different designer mistakes such as wrong state transitions, incorrect operators or incorrect module instantiations. The erroneous designs are then verified using industrial verification tools. A failure is detected and a counter-example is recorded and passed to the debugger. Experiments are conducted with three different versions of the SAT solver, the original MINISAT (**dbg-dom**), our enhanced version without the randomization heuristic after restarts (**dbg-dom+rev**), and our enhanced version with error-select variable order randomization at restarts (**dbg-dom+rev+RR**). Note that solution implications are applied in all experiments.

Table I shows the results of all our experiments. The first column gives the instance name. The next four columns respectively show the length of the counter example k, the number of nodes $|l|$ in C, the number of blocks $|B|$, and the number of solutions, *# sols*. Column **dbg-dom** gives the total run-time of the original MINISAT 2.2.0. Columns seven (*time*), eight (*# impl non-sols*) and nine (*imprv*) under **dbg-dom+rev** respectively give the total run-time of **dbg-dom+rev**, the number of implied non-solutions and the speed-up compared with **dbg-dom**. The following three columns show the same numbers for **dbg-dom+rev+RR**.

Figure 4 plots the ratio of implied non-solutions to all non-solutions using **dbg-dom+rev+RR** for each instance, sorted in increasing order. It can be seen that up to 75% of all non-solutions blocks are implied early and 25% of all non-solutions blocked are implied on average. As a result, the search-space of the SAT solver is pruned early, resulting in significant speed-ups.

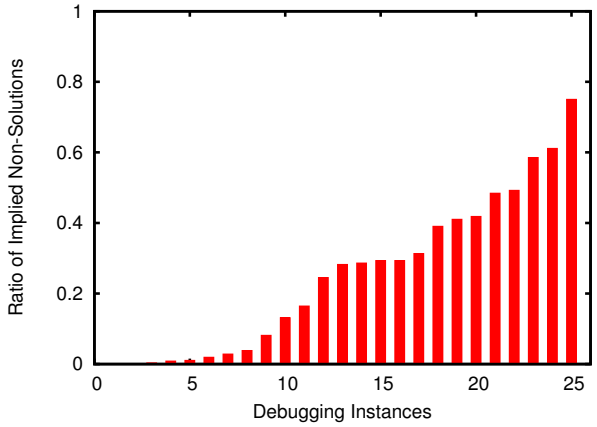

Fig. 4. Ratio of implied non-solutions to all non-solutions using **dbg-dom+rev+RR**

Figure 5 plots the number of solutions versus run-time for **dbg-dom** and **dbg-dom+rev+RR** for rsdecoder2. Clearly, **dbg-dom+rev+RR** outperforms **dbg-dom** by discovering solutions at a significantly faster rate. In addition to this faster rate, **dbg-dom+rev+RR** returns earlier solutions faster than its average rate (*i.e.,* its solutions plot is concave). This is beneficial because it allows the designer to examine those solutions earlier while the debugger continues to run.

Figure 6 plots the number of implied non-solutions versus run-time for **dbg-dom** and **dbg-dom+rev+RR** for rsdecoder2. In this figure, each implied non-solution found during the search is recorded at the time the SAT solver returns the next solution. Although **dbg-dom+rev** implies more non-solutions overall compared to **dbg-dom+rev+RR**, the latter learns non-solution blocks earlier. This is true in general, with slightly

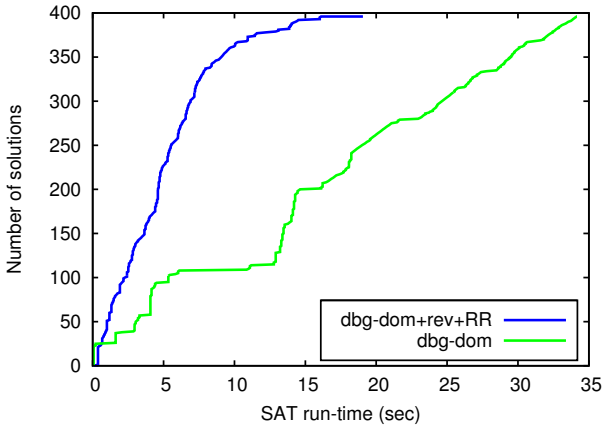

Fig. 5. # solutions vs run-time for rsdecoder2

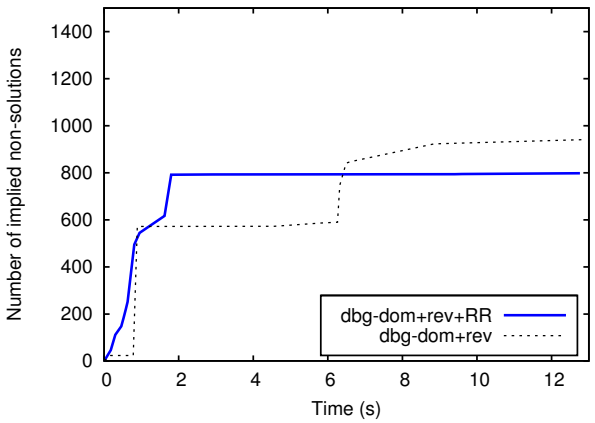

Fig. 6. # implied non-solutions vs run-time for rsdecoder2

more non-solutions being implied on average in **dbg-dom+rev** (28%) than in **dbg-dom+rev+RR** (25%). However, non-solution implications in **dbg-dom+rev+RR** are usually found earlier than in **dbg-dom+rev** because the former tries to actively go into parts of the search-space where it is easier to learn non-solutions. Returning non-solutions early is favorable because it helps the SAT solver prune the search-space faster.

The average speed-up in total SAT run-time compared to **dbg-dom** is 1.68x for **dbg-dom+rev** and 1.70x for **dbg-dom+rev+RR**, showing significant improvement. The difference between **dbg-dom+rev** and **dbg-dom+rev+RR** is small, however the latter is more consistent and shows improvement over **dbg-dom** in all cases. In some instances, such as for rsdecoder1, both versions of our solver terminate, while the original solver times out. In rare cases, such as ucrc_par and mem_ctrl2, no non-solutions are implied. However, our solvers still show significant speed-ups over **dbg-dom** due to our branching scheme which decides error-select variables first. Finally, Figure 7 plots the SAT run-times of our solvers **dbg-dom+rev** and **dbg-dom+rev+RR** versus those of **dbg-dom** on a logarithmic scale, demonstrating the effectiveness of our method.

VI. Conclusion

This work shows how to leverage reverse dominators in a circuit to speed-up SAT-based automated design debugging. This is done by performing non-solution implications, consisting of the early pruning of non-solution areas of the problem search-space. A new SAT branching strategy is also proposed for design debugging, which expedites the learning of non-solutions by the solver. Finally, an extensive set of experiments on real industrial designs demonstrates the robustness and practicality of the presented framework.

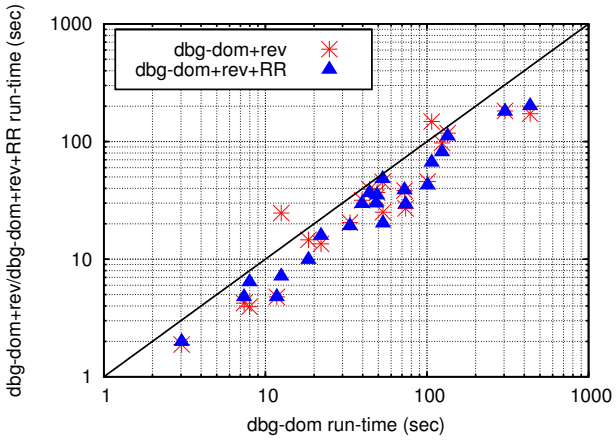

Fig. 7. Performance Results

References

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

[2] R. Reiter, "A theory of diagnosis from first principles," 1987.

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

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

[5] 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.

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

[7] B. Keng and A. Veneris, "Managing complexity in design debugging with sequential abstraction and refinement," in *ASP Design Automation Conf.*, 2011, pp. 479–484.

[8] A. Veneris, B. Keng, and S. Safarpour, "From RTL to silicon: the case for automated debug," in *ASP Design Automation Conf.*, 2011, pp. 306–310.

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

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

[11] R. Drechsler, *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[12] H. Mangassarian, A. Veneris, D. E.Smith, and S. Safarpour, "Debugging with dominance: On-the-fly debug solution implications," in *Int'l Conf. on CAD*, 2011.

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

[14] M. Ganai and A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," in *ASP Design Automation Conf.*, 2007, pp. 310–315.

[15] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic*. New York - London: Part 2. Consultants Bureau, 1968, pp. 115–125.

[16] 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.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

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

[19] OpenCores.org, "http://www.opencores.org," 2007.