

Debugging with Dominance: On-the-fly RTL Debug Solution Implications

Hratch Mangassarian, Andreas Veneris
University of Toronto
ECE Department, Toronto, ON M5S 3G4
{hratch,veneris}@eecg.toronto.edu

Duncan Exon Smith, Sean Safarpour
Vennsa Technologies, Inc.
Toronto, ON, M5V 3B1
{duncan,sean}@vennsa.com

ABSTRACT

Design debugging has become a resource-intensive bottleneck in modern VLSI CAD flows, consuming as much as 60% of the total verification effort. With typical design sizes exceeding the half-million synthesized gates mark, the growing number of blocks to be examined dramatically slows down the debugging process. The aim of this work is to prune the number of debugging iterations for finding all potential bugs, without affecting the debugging resolution. This is achieved by using structural dominance relationships between circuit components. More specifically, an iterative fixpoint algorithm is presented for finding dominance relationships between multiple-output blocks of the design. These relationships are then leveraged for the early discovery of potential bugs, along with their corrections, resulting in significant debugging speed-ups. Extensive experiments on real industrial designs show that 66% of solutions are discovered early due to dominator implications. This results in consistent performance gains in all cases and a 1.7x overall speed-up for finding all potential bugs, demonstrating the robustness and practicality of the proposed approach.

1. INTRODUCTION

Once functional verification discovers a discrepancy between a design and its specification, it returns a counter-example in the form of an error trace exhibiting an erroneous behavior of the design. Design debugging is the process of analyzing this counter-example and tracking down the bug(s) in the design. This is still a predominantly manual task in the industry. With the growing size and complexity of designs and error traces, bugs are increasingly difficult to locate. Hence, it comes as no surprise that today design debugging consumes as much as 60% of the total verification effort [1].

With the aim of alleviating the design debugging cost, several methodologies have been proposed over the years to automate this process [2–6]. The output of an 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 *corrections* can rectify the erroneous behavior in the given counter-example. The automated debugger must return all solutions, along with their corrections, with the engineer being given the final task of

identifying the real bug and fixing it.

Modern debuggers make heavy use of formal tools, such as Binary Decision Diagrams [3], Boolean Satisfiability (SAT) [4], Quantified Boolean Formulas [5] and Maximum Satisfiability [6]. In all these techniques, finding each solution requires a separate call to the formal engine. With typical design sizes exceeding the half-million synthesized gates mark, discovering solutions one-by-one is computationally expensive and limits the effectiveness of automated debuggers. This work addresses this issue by generating on-the-fly *implied* solutions, thus reducing the number of iterations for returning all solutions. This is done by using structural dominance relationships between circuit components.

A node u is said to be a (structural) *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 [7, 8] and have been used for optimizing various CAD tasks, *e.g.*, test pattern generation [9, 10]. More recently, they have been leveraged in the gate-level debugger in [4], which performs an initial debugging pass on selected dominator gates. However, state-of-the-art automated design debuggers operate on the RTL-level [11, 12], where bugs occur in *multiple-vertex, multiple-output blocks* in the circuit. As such, it is difficult to make use of single-vertex dominators at the RTL-level. A multiple-vertex block \mathbf{a} dominates another multiple-vertex block \mathbf{b} if every path from every node in \mathbf{b} to a primary output passes through a node in \mathbf{a} . Unlike existing approaches for finding multiple-vertex dominators, where block boundaries are not specified in advance [13–15], we are interested in establishing dominance relationships among a fixed set of blocks, naturally provided in a hierarchical RTL design.

The initial contribution of this work is a fixpoint algorithm that iteratively calculates dominance relationships between a predefined set of multiple-vertex blocks in a design. Next, it is proven that for each (set of) block(s) returned as a solution by the automated design debugger, every corresponding (set of) dominator(s) is a separate implied solution. As such, applying our fixpoint algorithm as a preprocessing step, the number of design debugging iterations for finding all solutions can be significantly reduced. Furthermore, we prove that corrections for implied solutions can be automatically generated without explicitly analyzing these solutions. It is shown that dominator-based solution implications are guaranteed to be valid given any error cardinality.

The proposed method is conveniently presented and implemented on top of a SAT-based automated design debugging framework [4, 11]. However, it is also applicable to simulation-based and other formal diagnosis techniques. 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 dominator implications. This results in a 1.7x overall speed-up in solving time in an industrial environment, demonstrating the robustness of the proposed approach.

The paper is organized as follows. Section 2 contains preliminaries on automated design debugging and dominators. Section 3 presents the iterative fixpoint algorithm for computing dominance relationships between blocks. Section 4 shows how to leverage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

block dominators for early solution implications in design debugging. Section 5 gives experimental results and Section 6 concludes the paper.

2. PRELIMINARIES

The following notation is used throughout the paper. Given a sequential circuit \mathcal{C} , the symbol l denotes the set of all nodes in \mathcal{C} . The symbols x , y and s label (possibly overlapping) subsets of l , respectively referring to the sets of primary inputs, primary outputs and state elements (flip-flops) of \mathcal{C} . For each $z \in \{x, y, s, l\}$, the Boolean variable z_i denotes the i th element in the set z .

To simplify the presentation, we consider designs with single clock-domains, although the described theory is applicable to multiple clock-domains [16]. *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 z), symbol z_i^t (or 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 t th clock-cycle is formalized using the transition relation predicate $T(s^t, s^{t+1}, x^t, y^t)$, which describes the dependence of the primary outputs y^t and next-state s^{t+1} on the primary inputs x^t and current-state 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 l^t as auxiliary variables.

The 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 $V = l \cup \{r\}$ and the set of edges $E = \{(l_i, l_j) | l_i \text{ is a fanin of } l_j \text{ in } \mathcal{C}\} \cup \{(y_i, r) | y_i \in y\}$. We reserve the letters u and v to refer to nodes in V . Let $\text{fanout}(v) = \{u \in V | (v, u) \in E\}$ and $\text{fanin}(v) = \{u \in V | (u, v) \in E\}$. Furthermore, let the nodes l of \mathcal{C} be 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 $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_{|\mathcal{B}|}\}$ denote the set of all blocks, where each $\mathbf{b}_i \subseteq l$ is a collection of nodes. Note that the same node l_i can belong to more than one block because of the hierarchical nature of RTL. The set $\text{out}(\mathbf{b}_i)$ denotes the outputs of block \mathbf{b}_i . In the unrolled circuit, the set \mathbf{b}_i^t ($\text{out}(\mathbf{b}_i^t)$) contains the (output) nodes of block \mathbf{b}_i in time-frame t . Finally, for each node v , we let $\text{out}^{-1}(v) = \{\mathbf{b}_j | v \in \text{out}(\mathbf{b}_j)\}$ denote the set of blocks in which v is an output.

Consider the sequential circuit in Figure 1(a). The blocks $\{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$ are shown in dotted boxes. We have $\text{out}(\mathbf{b}_1) = \{x_1\}$, $\text{out}(\mathbf{b}_3) = \{g_1, g_2\}$ and $\text{out}(\mathbf{b}_4) = \{g_3\}$. Furthermore, $\text{out}^{-1}(g_3) = \{\mathbf{b}_4\}$, $\text{out}^{-1}(s_1) = \{\mathbf{b}_5\}$, $\text{out}^{-1}(g_1) = \{\mathbf{b}_3\}$ and $\text{out}^{-1}(r) = \emptyset$. 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 .

2.1 Single-Vortex 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

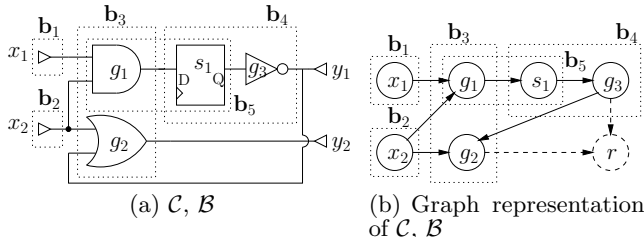


Figure 1: A sequential circuit with blocks

from v to the sink r passes through u . The set $\text{dom}(v) = \{u \in V | u \text{ dominates } v\}$ 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 $\text{idom}(v)$, is a provably unique node u ($u \neq v$) that dominates v and is dominated by all the nodes in $\text{dom}(v) - \{v\}$. It can be shown that for all $v \in V - \{r\}$, $\text{dom}(v) = \{v\} \cup \text{idom}(v) \cup \text{idom}(\text{idom}(v)) \cup \dots \cup \{r\}$ [17]. Therefore it is sufficient to compute all immediate dominators, which can be done in $O(|E| + |V|)$ time [7, 8]. In the directed graph shown in Figure 1(b), $\text{dom}(x_1) = \{x_1, g_1, s_1, g_3, r\}$, $\text{dom}(x_2) = \{x_2, r\}$, $\text{idom}(x_1) = \{g_1\}$, $\text{idom}(x_2) = \{r\}$.

In this work, we are interested in finding dominance relationships between blocks in \mathcal{B} , rather than between nodes in V . Section 3 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.

2.2 Design Debugging

This section describes SAT-based design debugging and introduces relevant notation, which is used throughout the paper. Given an erroneous design, a counter-example and an *error cardinality* N , the task of an automated design debugger is to find all sets of N blocks that can potentially be responsible for the counter-example. More precisely, each returned set of N blocks $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$, where $\{i_1, \dots, i_N\} \subseteq \{1, \dots, |\mathcal{B}|\}$, 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 . These solutions help manage the tremendous debugging complexity of modern designs [18] by significantly limiting the potentially buggy lines in the RTL. SAT-based automated design debugging [4, 11] encodes the debugging problem as a propositional formula whose satisfying assignments correspond to debugging solutions. The encoding process consists of several steps. Figure 2 illustrates a design debugging encoding for the circuit in Figure 1(a) and a two-cycle counter-example.

First, a set of *error-select* variables $e = \{e_1, \dots, e_{|\mathcal{B}|}\}$ are added to the circuit, such that setting $e_i = 1$ disconnects gates in $\text{out}(\mathbf{b}_i)$ 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 $\text{out}(\mathbf{b}_i^t)$ 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 $\text{out}(\mathbf{b}_i^t)$, 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

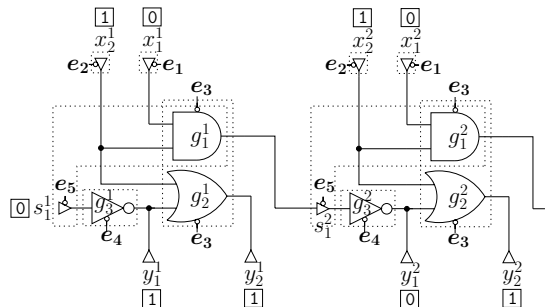


Figure 2: Design debugging formulation

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

$$Debug = \bigwedge_{t=1}^k T_{en}(s^t, s^{t+1}, x^t, y^t, e) \wedge \Phi_S(s^0) \wedge \Phi_X(x^1, \dots, x^k) \wedge \Phi_Y(y^1, \dots, y^k) \wedge \Phi_N(e) \quad (1)$$

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

Each assignment to $e = \{e_1, \dots, e_{|\mathcal{B}|}\}$ 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. In a satisfying assignment where some $e_i = 1$, the values of $out(\mathbf{b}_i^t)$ across all time-frames t represent a sequence of *corrections*, which fix the erroneous behavior in the counter-example. Note that *Debug* (1) allows these corrections to be non-deterministic functions of the applied primary inputs.

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 $\langle x_1, x_2 \rangle = \langle \langle 0, 1 \rangle, \langle 0, 1 \rangle \rangle$ and expected outputs $\langle y_1, y_2 \rangle = \langle \langle 1, 1 \rangle, \langle 0, 1 \rangle \rangle$, demonstrating a mismatch in the second time-frame at the output y_1 .

The corresponding design debugging formulation is illustrated in Figure 2. The constraints $\Phi_S = \bar{s}_1^1$, $\Phi_X = \bar{x}_1^1 x_2^1 x_1^2 x_2^2$ and $\Phi_Y = y_1^1 y_2^1 \bar{y}_1^2 y_2^2$ are shown in boxes, while Φ_N is omitted for brevity. For $N = 1$, $\{\mathbf{b}_1\}$, $\{\mathbf{b}_3\}$, $\{\mathbf{b}_4\}$ and $\{\mathbf{b}_5\}$ will be returned by the solver as separate solutions, and can therefore be considered potentially buggy blocks. Corrections for solution $\{\mathbf{b}_1\}$ (respectively $\{\mathbf{b}_3\}$, $\{\mathbf{b}_4\}$, $\{\mathbf{b}_5\}$) consist of the satisfying assignments to $\{x_1\}$ (respectively $\{g_1, g_2\}$, $\{g_3\}$, $\{s_1\}$) during the two time-frames. For instance, in any correction for solution $\{\mathbf{b}_1\}$, x_1^1 must be set to 1, whereas x_1^2 is a don't-care.

3. DOMINANCE BETWEEN BLOCKS

In this section, an iterative fixpoint algorithm is presented for finding all dominance relationships among a fixed set of multiple-vertex blocks, which are naturally defined in a hierarchical RTL design.

Definition 1 A block \mathbf{b}_j dominates another block \mathbf{b}_i , denoted as $\mathbf{b}_j \mathbf{D} \mathbf{b}_i$, if and only if every path from every node in \mathbf{b}_i to a primary output in 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 $\mathbf{D} \subseteq \mathcal{B} \times \mathcal{B}$ can be formalized using restricted quantifier notation [19] as follows:

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

where a path $p : v \xrightarrow{p} r$ 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 p from v to r , there exists a vertex u in p , such that $u \in out(\mathbf{b}_j)$ ”.

We let the set $\mathbf{D}(\mathbf{b}_i) = \{\mathbf{b}_j | \mathbf{b}_j \mathbf{D} \mathbf{b}_i\}$ consist of blocks that dominate \mathbf{b}_i . Note that $\mathbf{b}_i \mathbf{D} \mathbf{b}_i$ 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 \mathbf{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 [13–15], 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 [13, 14]. This is not applicable in a design debugging setting, where circuit blocks are defined in advance by the hierarchical RTL design.

In this work, the block dominator relation \mathbf{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 \mathbf{D} .

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

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

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

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

Algorithm 1 shows our pseudocode for computing the block dominator relation \mathbf{D} . It first computes the sets $\mathbf{d}(v)$ for every $v \in V$ (lines 1 to 21). This is done using a fixpoint 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 23 to compute $\mathbf{D}(\mathbf{b}_i)$ 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 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 to reach a fixpoint when computing the sets $\mathbf{d}(v)$ later in the algorithm.

Lines 3 to 6 calculate the sets $out^{-1}(v)$ for each node v . The iterative fixpoint algorithm for computing the sets $\mathbf{d}(v)$ for all nodes v (lines 8 to 20) is based on the traditional data-flow analysis algorithm for finding single-vertex dominators [17, 20]. Lines 8 and 9 initialize each dominator set $\mathbf{d}(v)$ 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 14. The computation of this refined set of dominator blocks of each node on line 14 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 $\mathbf{d}(v)$ are changed during an iteration (*i.e.*, the **if** condition on

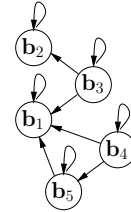


Figure 3: Block dominator relation \mathbf{D} of \mathcal{C}

Algorithm 1: Compute Block Dominators

```
input : Directed graph  $\mathcal{C}$ , blocks  $\mathcal{B}$ 
output: Block dominator relation  $\mathbf{D}$ 

1  $V \leftarrow \text{reversePostordering}(\mathcal{C}^T, r)$ ;
2 // For each node  $v$ , compute  $\text{out}^{-1}[v]$ : the set
   of blocks in which  $v$  is an output
3 foreach  $v \in V$  do  $\text{out}^{-1}[v] \leftarrow \emptyset$ ;
4 foreach  $\mathbf{b}_i \in \mathbf{b}$  do
5   | foreach  $v \in \text{out}[\mathbf{b}_i]$  do  $\text{out}^{-1}[v] \leftarrow \text{out}^{-1}[v] \cup \mathbf{b}_i$ ;
6 end
7 // Compute block-to-node dominator relation  $\mathbf{d}$ 
8  $\mathbf{d}[r] \leftarrow \emptyset$ ;
9 foreach  $v \in V - \{r\}$  do  $\mathbf{d}[v] \leftarrow \mathbf{B}$ ;
10  $\text{changed} \leftarrow \text{true}$ ;
11 while  $\text{changed}$  do
12   |  $\text{changed} \leftarrow \text{false}$ ;
13   | foreach  $u \in V$  in reverse postorder do
14     |  $\text{blocks} \leftarrow \bigcap_{\forall v \in \text{fanout}[u]} (\mathbf{d}[v] \cup \text{out}^{-1}[v])$ ;
15     | if  $\text{blocks} \neq \mathbf{d}[u]$  then
16       |  $\mathbf{d}[u] \leftarrow \text{blocks}$ ;
17       |  $\text{changed} \leftarrow \text{true}$ ;
18     | end
19   | end
20 end
21 foreach  $v \in V$  do  $\mathbf{d}[v] \leftarrow \mathbf{d}[v] \cup \text{out}^{-1}[v]$ ;
22 // Compute block dominator relation  $\mathbf{D}$ 
23 foreach  $\mathbf{b}_i \in \mathbf{B}$  do  $\mathbf{D}[\mathbf{b}_i] \leftarrow \bigcap_{\forall v \in \text{out}[\mathbf{b}_i]} \mathbf{d}[v]$ ;
```

line 15 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 21 adds the blocks in which node v is an output, to the dominators of v . Finally, on line 23, the block dominators $\mathbf{D}(\mathbf{b}_i)$ of each block \mathbf{b}_i are computed by intersecting the block dominators of each node in $\text{out}(\mathbf{b}_i)$.

Lemma 1 *The while loop in Algorithm 1 terminates and the block-to-node dominator relation \mathbf{d} is correctly computed by the end of the foreach loop on line 21.*

PROOF. In [21], the authors describe a class of iterative *data-flow analysis* algorithms. They use a very general lattice theoretic framework to analyze the termination and computation of this class of algorithms, which have a variety of applications (*e.g.*, in compiler optimization [22]) and are not restricted to calculating dominators. We will use the conclusions of [21] to analyze the computation of the block-to-node dominator relation \mathbf{d} in Algorithm 1.

Due to lack of space, we will avoid using lattice algebra, and will instead present the relevant results of [21] in our specific context. The class of algorithms described in [21] have a common structure, essentially conforming to lines 8 to 20 of Algorithm 1, but such that line 14 is replaced by:

$$\text{blocks} \leftarrow \bigcap_{\forall v \in \text{fanout}(u)} f_v(\mathbf{d}(v)) \quad (4)$$

with certain conditions specifying the types of functions f_v that are allowed. In this proof, we will show that using $f_v(\mathbf{d}(v)) = \mathbf{d}(v) \cup \text{out}^{-1}(v)$ (as done in Algorithm 1) satisfies the conditions put forth in [21], in order to prove the termination and correctness of our own algorithm.

Let F refer to any set of functions mapping sets of blocks to sets of blocks. Formally, F refers to a set of functions f of the

form:

$$\begin{aligned} f &: \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{P}(\mathcal{B}) \\ \mathbf{B} &\mapsto f(\mathbf{B}) \end{aligned} \quad (5)$$

where $\mathcal{P}(\mathcal{B})$ refers to the power set of \mathcal{B} (*i.e.*, the set of all subsets of \mathcal{B}) and $\mathbf{B} \subseteq \mathcal{B}$ is any arbitrary set of blocks. In [21], a set of such functions F is said to be *admissible* if and only if the following four conditions are satisfied:

1. All functions in F are distributive over \cup :
 $\forall \mathbf{B}, \mathbf{B}' \subseteq \mathcal{B}. \forall f \in F. (f(\mathbf{B} \cup \mathbf{B}') = f(\mathbf{B}) \cup f(\mathbf{B}'))$
2. F has an identity function:
 $\exists e \in F. \forall \mathbf{B} \subseteq \mathcal{B}. (e(\mathbf{B}) = \mathbf{B})$
3. F is closed under composition:
 $\forall f, g \in F. (f \circ g \in F)$
4. $\forall \mathbf{B} \subseteq \mathcal{B}. \exists H \subseteq F. (\mathbf{B} = \bigcap_{f \in H} f(\emptyset))$

Given a set of such admissible functions F and a directed graph $\mathcal{C} = (V, E, r)$ with output sink r , the authors of [21] map each vertex $v \in V$ to some function in F , which they call f_v . This mapping does not have to be one-to-one (*i.e.*, each f_v is not necessarily unique) or onto (*i.e.*, $\{f_v \mid \forall v \in V\} \subseteq F$). They prove that if F is admissible, then any algorithm that conforms to lines 8 to 20 in Algorithm 1, with line 14 replaced by (4), terminates. Furthermore, they show that in such a scenario, at the completion of this **while** loop, for each $v \in V$, we get:

$$\mathbf{d}(v) = \bigcap_{\substack{\forall \text{ paths } p=(v, v_1, v_2, \dots, v_n, r) \\ \text{such that } v \xrightarrow{p} r}} f_{v_1}(f_{v_2}(\dots f_{v_n}(f_r(\emptyset)) \dots)) \quad (6)$$

In our case, we use the following set of admissible functions:

$$F^* = \{f^*(\mathbf{B}) = \mathbf{B} \cup \mathbf{B}' \mid \forall \mathbf{B}' \subseteq \mathcal{B}\}.$$

We leave it to the reader to verify that F^* is indeed admissible (*i.e.*, it satisfies the four conditions given above), due to lack of space. Next, as done in [21], we map each node $v \in V$ to some function $f_v^* \in F^*$, where:

$$f_v^*(\mathbf{B}) = \mathbf{B} \cup \text{out}^{-1}(v).$$

Clearly, $\{f_v^*(\mathbf{B}) = \mathbf{B} \cup \text{out}^{-1}(v) \mid \forall v \in V\} \subseteq F^*$.

Replacing the functions f_v^* in (4) yields line 14 in our algorithm. Since f_v^* 's are drawn from the admissible set of functions F^* , the **while** loop in Algorithm 1 terminates. Furthermore, using (6), at the completion of this **while** loop, we have:

$$\mathbf{d}(v) = \bigcap_{\forall p[v \xrightarrow{p} r]} \left(\bigcup_{\forall u \in p - \{v\}} \text{out}^{-1}(u) \right) \quad (7)$$

Finally, on line 21, $\text{out}^{-1}(v)$ is added to each $\mathbf{d}(v)$. As such, by the end of the **foreach** loop on line 21, we have:

$$\begin{aligned} \mathbf{d}(v) &= \left(\bigcap_{\forall p[v \xrightarrow{p} r]} \left(\bigcup_{\forall u \in p - \{v\}} \text{out}^{-1}(u) \right) \right) \cup \text{out}^{-1}(v) \\ &= \bigcap_{\forall p[v \xrightarrow{p} r]} \left(\bigcup_{\forall u \in p} \text{out}^{-1}(u) \right) = \bigcap_{\forall p[v \xrightarrow{p} r]} \left(\bigcup_{\forall u \in p} \{\mathbf{b}_j \mid u \in \text{out}(\mathbf{b}_j)\} \right) \\ &= \bigcap_{\forall p[v \xrightarrow{p} r]} \{\mathbf{b}_j \mid \exists u \in p. (u \in \text{out}(\mathbf{b}_j))\} \\ &= \{\mathbf{b}_j \mid \forall p[v \xrightarrow{p} r]. \exists u \in p. (u \in \text{out}(\mathbf{b}_j))\} \end{aligned}$$

As such, the computed sets $\mathbf{d}(v)$ satisfy the definition of the block-to-node dominator relation \mathbf{d} given in (3). \square

Theorem 1 *Algorithm 1 correctly computes the block dominator relation \mathbf{D} .*

PROOF. $D(\mathbf{b}_i)$ is computed on line 23 as $\bigcap_{v \in \text{out}(\mathbf{b}_i)} d(v)$. Using Lemma 1, we get:

$$\begin{aligned} D(\mathbf{b}_i) &= \bigcap_{v \in \text{out}(\mathbf{b}_i)} \{ \mathbf{b}_j | \forall p[v \xrightarrow{p} r]. \exists u[u \in p]. (u \in \text{out}(\mathbf{b}_j)) \} \\ &= \{ \mathbf{b}_j | \forall v[v \in \text{out}(\mathbf{b}_i)]. \forall p[v \xrightarrow{p} r]. \exists u[u \in p]. (u \in \text{out}(\mathbf{b}_j)) \} \end{aligned}$$

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

The overall run-time of Algorithm 1 is normally dictated by the run-time of the **while** loop from line 11 to 20. Furthermore, during each iteration of the **while** loop, line 14 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 14 during a single iteration of the **while** loop, it can be seen that line 14 performs a total of $O(|E|)$ intersections and unions between two sets of size at most $|\mathcal{B}|$ (since $d(v), \text{out}^{-1}(v) \subseteq \mathcal{B}$). 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 14 takes $O(|\mathcal{B}| \cdot |E|)$ time.

Let c denote the *loop-connectedness* of the directed graph \mathcal{C} , 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 Depth-First Search performed in `reversePostordering`(\mathcal{C}^T, r) on line 1. It is proven in [21] that the number of iterations of the **while** loop for the general class of such fixpoint algorithms is bounded by $c + 2$, if and only if the following condition holds:

$$\forall f, g[f, g \in F]. ((f \circ g)(\emptyset) \supseteq g(\emptyset) \cap f(\mathcal{B})) \quad (8)$$

We show that (8) holds for our set of admissible functions F^* given in the proof of Lemma 1. Consider any two functions $f^*, g^* \in F^*$ such that $f^*(\mathbf{B}) = \mathbf{B} \cup \mathbf{B}'$ and $g^*(\mathbf{B}) = \mathbf{B} \cup \mathbf{B}''$, where $\mathbf{B}', \mathbf{B}'' \subseteq \mathcal{B}$ are arbitrary sets of blocks. We have:

$$(f^* \circ g^*)(\emptyset) = f^*(g^*(\emptyset)) = f^*(\mathbf{B}'') = \mathbf{B}' \cup \mathbf{B}''$$

and

$$g^*(\emptyset) \cap f^*(\mathcal{B}) = \mathbf{B}'' \cap \mathcal{B} = \mathbf{B}''$$

clearly satisfying (8). Therefore, our fixpoint algorithm takes $O(c \cdot |\mathcal{B}| \cdot |E|)$ time.

4. LEVERAGING BLOCK DOMINANCE IN DESIGN DEBUGGING

In this section, we show how to leverage the relation D to imply solutions early in the design debugging iterations. 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 $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$, every set of the form $\{\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}\}$ such that $(\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}) \in D(\mathbf{b}_{i_1}) \times \dots \times D(\mathbf{b}_{i_N})$ is also a solution of *Debug* (1). Furthermore, it is shown that corrections for each implied solution can also be obtained automatically from the satisfying assignment of the original solution.

First, due to the fixed length of a given counter-example, we must 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 $\mathbf{b}_j D_k \mathbf{b}_i$, if and only if every path containing at most k state elements, starting from every node in $\text{out}(\mathbf{b}_i)$ to r passes through a node in $\text{out}(\mathbf{b}_j)$.

The following three Lemmas are used in the proof of Theorem 2. Note that in Lemma 2, $\bigcup_{n=1}^N \mathbf{b}_{i_n}$ (respectively $\bigcup_{n=1}^N \mathbf{b}_{j_n}$) denotes a “super-block” consisting of all nodes in blocks $\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}$ (respectively $\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}$).

Lemma 2 $\bigwedge_{n=1}^N (\mathbf{b}_{j_n} D \mathbf{b}_{i_n}) \Rightarrow \left(\bigcup_{n=1}^N \mathbf{b}_{j_n} \right) D \left(\bigcup_{n=1}^N \mathbf{b}_{i_n} \right)$

PROOF. If $\forall n[1 \leq n \leq N]$, any path from any node in $\text{out}(\mathbf{b}_{i_n})$ to a primary output passes through a node in $\text{out}(\mathbf{b}_{j_n})$, then clearly any path from any node in one of $\text{out}(\mathbf{b}_{i_1}), \dots, \text{out}(\mathbf{b}_{i_N})$ to a primary output passes through a node in one of $\text{out}(\mathbf{b}_{j_1}), \dots, \text{out}(\mathbf{b}_{j_N})$. \square

Lemma 3 $\mathbf{b}_j D \mathbf{b}_i \Rightarrow \mathbf{b}_j D_k \mathbf{b}_i$

PROOF. If $\mathbf{b}_j D \mathbf{b}_i$ 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 . \square

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

PROOF. True by construction. \square

Theorem 2 Given an erroneous design \mathcal{C} , a counter-example of length k along with the corresponding expected outputs and an error cardinality N , if $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$ is a solution of *Debug* (1) and $\bigwedge_{n=1}^N (\mathbf{b}_{j_n} D \mathbf{b}_{i_n})$, then $\{\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}\}$ is an implied solution of *Debug* (1).

PROOF. The theorem can be formalized as:

$$\text{Debug} \wedge \bigwedge_{n=1}^N e_{i_n} \text{ is SAT} \Rightarrow \text{Debug} \wedge \bigwedge_{n=1}^N e_{j_n} \text{ is SAT} \quad (9)$$

where we refer to the left-hand-side (right-hand-side) formula of the implication as the LHS (RHS).

Let \mathcal{U} refer to the k -time-frame expanded circuit obtained from \mathcal{C} as described in Subsection 2.2. 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 blocks $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$ (respectively $\{\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}\}$) across all time-frames in \mathcal{U} . Also, let $\text{out}(I)$ (respectively $\text{out}(J)$) 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 $\text{out}(J)$ in \mathcal{U} . Let \mathcal{U}_I denote the nodes in \mathcal{U} that are in the transitive fanout of $\text{out}(I)$, 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 . We know that $\bigwedge_{n=1}^N (\mathbf{b}_{j_n} D \mathbf{b}_{i_n})$, and by Lemma 2 and Lemma 3, we get $\left(\bigcup_{n=1}^N \mathbf{b}_{j_n} \right) D_k \left(\bigcup_{n=1}^N \mathbf{b}_{i_n} \right)$. Given this and Lemma 4, we can imply that any path from $\text{out}(I)$ to a primary output must pass through $\text{out}(J)$. As a result, these partitions of \mathcal{U} can be represented by the diagram shown in Figure 4.

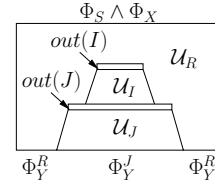


Figure 4: Partition of \mathcal{U}

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 $e_{i_1} = 1, \dots, e_{i_N} = 1$, 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 $e_{j_1} = 1, \dots, e_{j_N} = 1$, 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 $e_{j_1} = 1, \dots, e_{j_N} = 1$ 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. \square

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

PROOF. In the proof of Theorem 2, we showed how to build a satisfying assignment π' of the RHS of (9) 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 $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$, a sequence of corrections for the implied solution $\{\mathbf{b}_{j_1}, \dots, \mathbf{b}_{j_N}\}$ simply consists of the assignments in π to $out(J) = \{out(\mathbf{b}_{j_n}^t) | 1 \leq n \leq N, 1 \leq t \leq k\}$. \square

4.1 Overall Flow

The flowchart in Figure 5 illustrates the overall design debugging flow using on-the-fly dominator implications. Algorithm 1 is first run to compute $D(\mathbf{b}_i)$ for every block $\mathbf{b}_i \in \mathcal{B}$. Next, the automated debugger builds the original debugging problem, Debug (1), and passes it to the SAT solver. If it is UNSAT, the flow terminates. Otherwise, a solution $\{\mathbf{b}_{i_1}, \dots, \mathbf{b}_{i_N}\}$ 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.

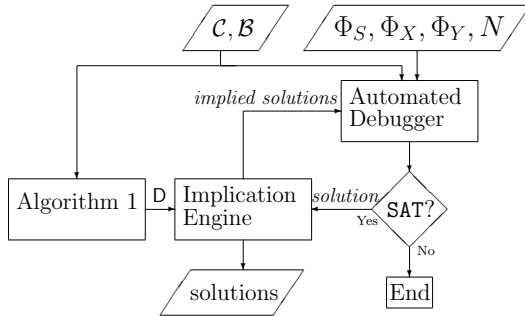


Figure 5: Debugging using Dominance Flow Chart

Example 2 *Consider the sequential circuit in Figure 1(a) and the corresponding design debugging formulation illustrated in Figure 2. Assume that $D \subseteq \mathcal{B} \times \mathcal{B}$ has been computed using Algorithm 1. Furthermore, assume that $N = 1$, and that the solver first returns the solution $\{\mathbf{b}_1\}$. Since $D(\mathbf{b}_1) = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$, the solutions $\{\mathbf{b}_3\}$, $\{\mathbf{b}_4\}$ and $\{\mathbf{b}_5\}$ (along with their corrections) can be immediately implied, eliminating three SAT iterations for finding these solutions. After adding the corresponding blocking clauses (\bar{e}_1) , (\bar{e}_3) , (\bar{e}_4) and (\bar{e}_5) to Debug, the solver returns UNSAT, indicating that all solutions have been found.*

5. EXPERIMENTAL RESULTS

This section presents the experimental results for the proposed dominator-based design debugging flow. All experiments are run using a single core of a Core 2 Quad 2.66 Ghz workstation with 8 GB of RAM and a timeout of 3600 seconds. The proposed debugging framework is implemented using a state-of-the-art hierarchical SAT-based debugger based on [4, 11], with a Verilog front-end to allow for RTL diagnosis. MINISAT-v2.2 [23] is used to solve all SAT instances.

Seven industrial Verilog RTL designs from OpenCores [24] and three 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. 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 state-

Table 1: Instance Information

Instance	$ I $	$ \mathcal{B} $	k	N
fdct-1	365 574	4 665	142	1
fdct-2	365 574	4 666	146	1
mips789-1	63 241	2 750	24	1
mips789-2	30 171	876	68	1
mips789-3	30 711	904	153	2
usb_funct-1	35 158	3 397	31	1
usb_funct_2	36 181	3 477	25	1
usb_funct_3	36 181	3 401	36	2
wb_dma-1	301 812	8 460	20	1
wb_dma-2	187 874	6 236	69	1
fpu-1	79 504	1 988	312	1
fpu-2	139 932	2 145	312	1
opensparc_ddr2-1	64 915	2 777	23	1
opensparc_ddr2-2	58 399	2 779	31	1
vga-1	89 402	1 741	11 308	1
vga-2	89 488	1 833	7	1
vga-3	89 488	1 741	508	2
design1-1	242 086	16 736	25	1
design1-2	532 610	51 564	27	1
design1-3	203 718	10 258	151	1
design1-4	203 706	10 246	5	1
design1-5	532 634	51 564	29	1
design1-6	690 766	51 564	27	1
design2-1	875 837	84 975	212	1
design2-2	875 837	84 975	212	1
design2-3	875 837	84 975	212	1
design3-1	499 705	20 211	562	1
design3-2	499 705	20 211	177	1
design3-3	499 705	20 211	252	2

Table 2: Debugging with and without Dominance

Instance	Common		dbg-trad	dbg-dom						
	overhead (sec)	total # sols	dbg (sec)	avg D	# impl	% impl	dom (sec)	dbg (sec)	dom+dbg (sec)	impr (x)
fdct-1	52.4	79	95.1	15.2	52	66%	7.8	37.4	45.2	2.1x/1.5x
fdct-2	52.2	93	188.0	15.2	64	69%	7.8	148.2	156.0	1.2x/1.2x
mips789-1	15.7	162	76.1	14.2	100	62%	1.0	33.4	34.4	2.2x/1.8x
mips789-2	18.1	37	30.4	9.0	21	57%	0.8	18.5	19.3	1.6x/1.3x
mips789-3	38.4	45	84.4	10.6	34	76%	0.6	52.2	52.8	1.6x/1.3x
usb_funct-1	11.9	423	163.5	10.6	231	55%	0.1	76.4	76.5	2.1x/2.0x
usb_funct-2	9.6	93	21.4	10.2	51	55%	0.9	11.8	12.7	1.7x/1.4x
usb_funct-3	14.6	3 517	1 667.1	10.6	2 656	76%	0.2	690.4	690.6	2.4x/2.4x
wb_dma-1	71.0	135	105.8	14.6	75	56%	5.0	50.8	55.8	1.9x/1.4x
wb_dma-2	32.0	234	178.6	20.8	125	53%	1.6	119.3	120.9	1.5x/1.4x
fpu-1	168.0	8	20.8	10.4	3	38%	1.6	16.0	17.6	1.2x/1.0x
fpu-2	18.4	80	23.5	7.0	43	54%	0.6	16.1	16.7	1.4x/1.2x
opensparc_ddd2-1	17.1	76	48.2	8.6	44	58%	0.2	20.7	20.9	2.3x/1.7x
opensparc_ddd2-2	11.5	99	34.9	8.6	56	57%	0.2	12.7	12.9	2.7x/1.9x
vga-1	72.0	23	50.2	19.3	21	91%	2.3	20.9	23.2	2.2x/1.3x
vga-2	2.9	52	2.2	19.3	28	54%	0.2	1.2	1.4	1.5x/1.2x
vga-3	11.0	1 226	941.8	19.3	1 088	89%	0.2	600.5	600.7	1.6x/1.6x
design1-1	94.0	93	240.6	17.9	53	57%	5.8	135.4	141.2	1.7x/1.4x
design1-2	188.2	122	1 214.0	32.4	93	76%	34.2	869.5	903.7	1.3x/1.3x
design1-3	36.3	127	119.8	18.8	85	67%	3.3	52.5	55.8	2.1x/1.7x
design1-4	16.5	41	31.2	22.5	27	66%	0.5	28.3	28.8	1.1x/1.1x
design1-5	174.5	58	832.3	32.3	45	78%	31.1	634.7	665.8	1.3x/1.2x
design1-6	219.0	71	1 978.3	15.7	40	56%	19.6	1 046.4	1 066.0	1.9x/1.7x
design2-1	456.3	40	410.0	22.7	27	68%	39.2	327.7	366.9	1.1x/1.1x
design2-2	472.7	42	312.2	22.7	32	76%	39.2	228.1	267.3	1.2x/1.1x
design2-3	454.8	32	313.1	22.8	21	66%	39.2	234.7	273.9	1.1x/1.1x
design3-1	99.6	117	180.9	48.8	88	75%	13.1	80.2	93.3	1.9x/1.5x
design3-2	84.8	89	127.0	48.8	67	75%	13.1	63.9	77.0	1.6x/1.3x
design3-3	83.4	1 120	2 326.5	48.8	953	85%	13.1	1 467.0	1 480.1	1.6x/1.5x

ment, a module definition, an instantiation, etc. Experiments are conducted with and without dominator implications. **dbg-trad** refers to the “traditional” debugging flow (without an implication engine), and **dbg-dom** refers to our extended debugging flow using dominator implications, illustrated in Figure 5.

Table 1 shows the circuit characteristics of each design debugging instance. The first column gives the instance name, which consists of the design name and an appended number indicating a different inserted error. The following four columns respectively show the number gates $|I|$, the number of blocks $|\mathcal{B}|$, the number of clock-cycles k in the counter-example, and finally the error cardinality N .

Table 2 shows the results of all our experiments. The first column gives the instance name. Columns *overhead* and *total #sols* respectively refer to the run-time overhead for setting up the problem (*i.e.*, generating the CNF of *Debug*) and the total number of returned solutions. The *overhead* run-time includes graph optimizations such as dangling logic removal. The *overhead* and the *total #sols* are common for both **dbg-trad** and **dbg-dom**. Note that the number of solutions for instances with $N = 2$ can be greater than \mathcal{B} (*e.g.*, *usb_funct-3*) because each two-block combination $\{b_{i_1}, b_{i_2}\}$ that can be modified to correct the counter-example is a solution.

Column four (*dbg*) shows the total SAT solver run-time using **dbg-trad** for finding all debugging solutions. The remaining columns present the results of our proposed framework, **dbg-dom**. Column *avg |D|* shows the average size of the sets $D(b_i)$ computed by Algorithm 1. Next, columns *#impl* and *%impl* respectively show the number of implied solutions among all solutions and the percentage of implied solutions among all solutions. Column *dom* shows the run-time of Algorithm 1 for computing the

block dominator relation D . Column *dbg* gives the total SAT solver run-time using **dbg-dom**, while column *dom+dbg* adds to this the dominator computation run-time of Algorithm 1. Finally, column *impr* shows the speed-up achieved by **dom+dbg** over **dbg-trad**, first excluding then including the common overhead.

Figure 6 plots the ratio of implied solutions for each instance, sorted in increasing order. On average, 66% of all solutions are implied. In other terms, the number of calls to the SAT solver is reduced by a factor of 2.9x due to the early discovery of solutions

Figure 6: Ratios of implied solutions to all solutions

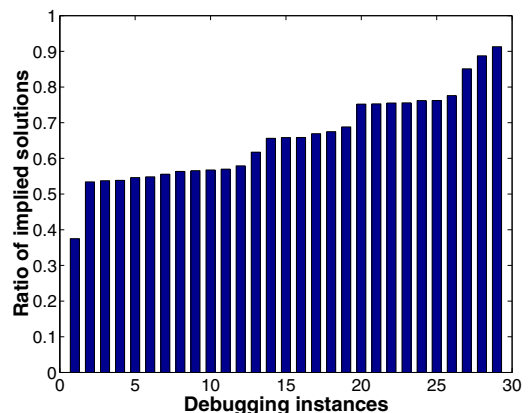
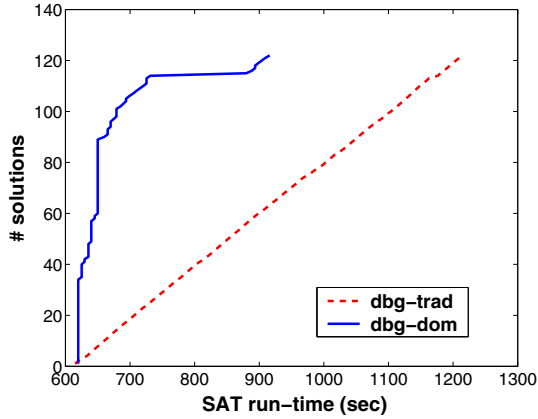


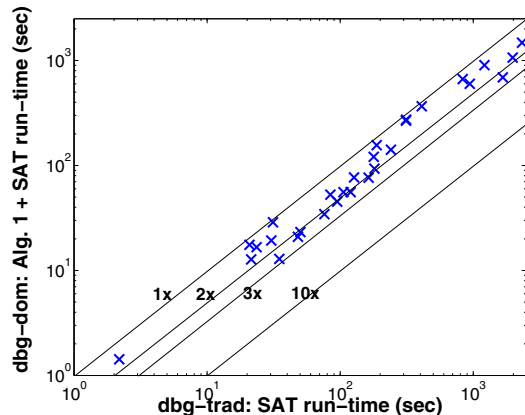
Figure 7: # solutions vs. run-time for design1-2



using our approach. For each solution found by the SAT solver, about 2.6 more solutions are implied on average. This number is significantly less than the average number of dominators of each block, which is 19.5, because many implied solutions in later iterations might have already been found (or implied) in previous iterations. Figure 7 plots the number of found solutions versus run-time for both **dbg-trad** and **dbg-dom** for design1-2. It can be seen that while **dbg-trad** returns solutions at roughly equal time intervals, **dbg-dom** initially discovers solutions at a fast rate due to new implications, but the rate of discovery of new solutions decreases with time. Returning most solutions early is beneficial because the designer can start examining returned solutions earlier, while the debugger continues to run.

The average speed-up in total SAT run-time from **dbg-trad** to **dbg-dom** is 1.8x. In many cases, higher percentages of implied solutions mean less debugging iterations, which result in less total SAT solving time. For instance, in vga-1, 21 out of 23 solutions (91%) are implied, yielding a 2.4x speed-up in total SAT run-time, compared to the averages of 66% implied solutions and a 1.8x 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 implications. Including the time to compute the dominator relation D , the speed-up from **dbg-trad** to **dbg-dom** is about 1.7x disregarding common overhead, and 1.4x including common overhead. Figure 8 plots the run-times of our approach ($dom+dbg$) versus those of **dbg-trad** on a logarithmic scale, along with the 1x, 2x, 3x and 10x lines, clearly showing the consistent superiority of the proposed method.

Figure 8: dbg-dom vs. dbg-trad run-time comparison



6. CONCLUSION

We present an iterative fixpoint algorithm for computing dominance relationships between multiple-output blocks of a design. We then show how to leverage these dominance relationships to reduce the number of design debugging iterations for finding all potential bug locations, or solutions, in a design. Furthermore, we prove that corrections for implied solutions can be automatically generated without explicitly analyzing these solutions. Finally, an extensive set of experiments on real industrial designs demonstrates the consistent benefits of the presented framework.

7. REFERENCES

- [1] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.
- [2] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.
- [3] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.
- [4] 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.
- [5] 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.
- [6] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Trans. on CAD*, vol. 29, pp. 1804–1817, November 2010.
- [7] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM J. Comput.*, vol. 28, no. 6, pp. 2117–2132, 1999.
- [8] L. Georgiadis and R. E. Tarjan, "Finding dominators revisited: extended abstract," in *SODA*, 2004, pp. 869–878.
- [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] 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.
- [12] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.
- [13] R. Gupta, "Generalized dominators and post-dominators," in *Symposium on Principles of Programming Languages*, 1992, pp. 246–257.
- [14] S. Alstrup, J. Clausen, and K. Jørgensen, "An $O(|V|*|E|)$ algorithm for finding immediate multiple-vertex dominators," *Inf. Process. Lett.*, vol. 59, no. 1, pp. 9–11, 1996.
- [15] R. Krenz and E. Dubrova, "A fast algorithm for finding common multiple-vertex dominators in circuit graphs," in *ASP Design Automation Conf.*, 2005, pp. 529–532.
- [16] M. Ganai and A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," in *ASP Design Automation Conf.*, 2007, pp. 310–315.
- [17] K. Cooper, T. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, pp. 1–10, 2001.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] J. Kam and J. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM*, vol. 23, no. 1, pp. 158–171, 1976.
- [22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [23] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.
- [24] OpenCores.org, "http://www.opencores.org," 2007.