Ann Math Artif Intell
https://doi.org/10.1007/s10472-018-9578-x

Efficient suspect selection in unreachable state diagnosis

Ryan Berryhill[1] · **Andreas Veneris**[1]

Abstract In the modern hardware design cycle, correcting the design when verification reveals a state to be erroneously unreachable can be a time-consuming manual process. Recently-developed algorithms aid the engineer in finding the root cause of the failure in these cases. However, they exhaustively examine every design location to determine a set of possible root causes, potentially requiring substantial runtime. This work develops a novel approach that is applicable to practical diagnosis problems. In contrast to previous approaches, it considers only a portion of the design locations but still finds the complete solution set to the problem. The presented approach proceeds through a series of iterations, each considering a strategically-chosen subset of the design locations (a suspect set) to determine if they are root causes. The results of each iteration inform the choice of suspect set for the next iteration. By choosing the first iteration's suspect set appropriately, the algorithm is able to find the complete solution set to the problem. Empirical results on industrial designs and standard benchmark designs demonstrate a 15x speedup compared to the previous approach, while considering only 18.7% of the design locations as suspects.

Keywords Diagnosis · Debug · Verification · Model checking

Mathematics Subject Classification (2010) 94C12 · 68W35

✉ Ryan Berryhill
ryan@eecg.toronto.edu

Andreas Veneris
veneris@eecg.toronto.edu

[1] Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario, M5S 3G4, Canada

Published online: 13 March 2018

1 Introduction

Functional verification has become the primary bottleneck in the modern design cycle, accounting for up to 70% of the design effort [10]. Debugging accounts for half of the time spent in verification [9]. Many verification and debugging tasks are automated or partially automated, somewhat mitigating this substantial cost. However, due to the computational resources they demand, some of these techniques can be difficult to apply in practice, particularly when considering large designs.

When verification reveals erroneous behavior in the form of an observation value mismatch, an error-trace is returned that demonstrates the problem. A traditional SAT-based debugging tool [17] can then be applied to diagnose the failure. Many techniques [11, 12, 14, 16] are used to increase the scalability of SAT-based debugging tools, allowing them to handle larger designs and error traces than would otherwise be possible.

Functional verification may also involve model checking, which determines if a given state is reachable. When model checking reveals that a state is unreachable in violation of the design specification, no error-trace is readily available. Debugging therefore requires a labor-intensive first step where the engineer must manually find a trace that should reach the desired state but instead reaches a different state. Recently, specialized SAT-based automated debugging tools [1, 2] have been developed to diagnose this type of error. The work of [1] involves determining an approximation of the set of reachable states which is used as part of the input to a traditional debugging tool. While useful to the engineer, this approach may not find the complete solution set to the problem. Conversely, the work of [2] is complete with respect to its input set of suspect locations. That is, given a set of suspect locations, it returns every location in that set where a change can be made to make the target state reachable (i.e., solutions). However, its runtime appears to increase sharply as more suspects are added, limiting its applicability in debugging large designs.

To address this limitation, this work presents a novel automated debugging framework that eliminates the need to specify a suspect set [3]. Given an unreachable target state, the algorithm returns the complete set of solutions to the problem. Unlike the approach of [2], it neither requires an input suspect set nor exhaustively considers every design location as a suspect. Instead, it proceeds through a series of iterations, each of which considers a suspect set that is small relative to the size of the circuit. The key innovation is that the non-solution suspects may indicate that other design locations are not solutions, and can be ignored in future iterations. In this manner, the algorithm is able to find the complete solution set while considering only a small number of suspects. Experimental results show that only 18.7% of the design locations are considered as suspects. Ultimately, this avoids the runtime explosion of the previous approach. The approach is complete under the assumption that the observed failure is caused by a single erroneous design location. In practice, diagnosis algorithms are typically used under this assumption anyway, meaning this limitation is of little concern.

In greater detail, the methodology operates as follows. The algorithm accepts a circuit represented by an And-Inverter graph (AIG) and a predicate representing an unreachable target state as input. It begins by constructing an initial suspect set that includes all registers that appear in the target state predicate. The initial suspect set also includes all locations in the design with multiple fanout, as these locations could otherwise be incorrectly ignored. An underlying debugging algorithm such as that of [2] is executed with this suspect set. Any solution returned by the debugging algorithm is recorded to later be returned to the user. Additionally, the fanin of any solution is added to the suspect set for the subsequent

iteration. Iterations are executed in this manner until an iteration finds no solutions. The solution set of each iteration tends to be small in practice, so most locations are ignored. Assuming the observed failure has a single root cause, all ignored locations are provably not solutions, so the algorithm remains complete under this assumption.

Experiments on industrial designs demonstrate the effectiveness of the proposed approach. It is compared against the debugging approach of [2], where the debugging algorithm is given a suspect set that includes every design location. In all cases, both approaches find the same solution set, but the proposed methodology is found to be an average of 15× faster than the previous approach, while ignoring 81.3% of design locations.

The remainder of this paper is organized as follows. Section 2 presents background material. Section 3 explains the debugging algorithm of [2], as it is important to the work presented here. Section 4 presents the efficient suspect selection algorithm. Section 5 presents experimental results demonstrating the benefits of the presented approach. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 Notation and terminology

The following notation and terminology is used throughout this work. Each assignment to the state elements of a sequential circuit C is referred to as a *state* of C. The transition relation of C is denoted by T. For a state pair $\langle t, t' \rangle$, $\langle t, t' \rangle \in T$ if and only if there exists an assignment to the primary inputs that causes C to transition from state t to state t'. The set of initial states of C is denoted I. For a predicate P over the state elements of C, any state $t \in P$ is a P-state. A sequence of states $t_0, ..., t_n$ is a *trace* of C if and only if $\langle t_i, t_{i+1} \rangle \in T$ for all $0 \leq i < n$ and $t_0 \in I$. A state t is *reachable* under C if it appears in a trace of C. It is also *i-step reachable* if it appears in a trace of i or fewer cycles.

A circuit can be represented by an And-Inverter graph (AIG) [6]. An AIG is a directed acyclic graph (DAG) in which each vertex represents either an AND-gate, a NOT-gate (inverter), an input, an output, or a sequential element (register). An AND-gate vertex has exactly two in-edges representing its inputs and one or more out-edges representing its output. Similarly, a NOT-gate vertex has a single in-edge representing its input and one or more out-edges representing its output. An input vertex has an in-degree of zero and one or more out-edges, while an output vertex has no out-edges and one in-edge. A register is represented by two vertices. The register's next-state input is a vertex with one in-edge representing the input signal and no out-edges. A register's output is a vertex with no in-edges and at least one out-edge representing the register's output signal.

In a circuit C, a *location* refers to any vertex of the AIG. For a location l, the set $fanin(l)$ refers to all locations in the fanin of l. In the AIG representation, $fanin(l)$ contains every location l' for which an edge (l', l) exists. Additionally, if l is the output of a register, $fanin(l)$ is the next-state input of the register. Similarly, the set $fanout(l)$ refers to all locations in the fanout of l. These relations are symmetric. If $l \in fanout(l')$, then $l' \in fanin(l)$ and vice versa. The cone-of-influence (COI) for a location is defined recursively in terms of i-step COIs as follows. The 1-step COI for a location l is the set of all locations l' for which the AIG contains a path from l' to l, as these are the set of all locations that can influence the value at l in a single clock cycle. The i-step COI for a location l is the set of all locations l' for which a path exists from l' to a register in the $(i-1)$-step COI. The COI for l is the set of locations that appear in any i-step COI of l.

2.2 Circuit verification with boolean satisfiability

Circuit verification often involves proving that a particular condition can never occur. For instance, it may be of interest to prove that a given circuit can never produce invalid output. Boolean Satisfiability (SAT) is well-suited to this sort of problem. A combinational circuit can be converted into a Boolean formula in Conjunctive Normal Form (CNF) in linear time [18]. Additional constraints can be added to produce a formula that is satisfiable if and only if the circuit can produce some particular output. The formula is then given to a SAT solver. If it is unsatisfiable, no input to the circuit causes it to produce the output in question. Alternatively, if the formula is satisfiable, then the satisfying assignment indicates an input assignment that produces the relevant output.

Often, verification problems are concerned with the sequential behavior of the circuit. Sequential behavior can be modeled for a fixed number of clock cycles using a CNF formula. This is accomplished using an Iterative Logic Array (ILA) [13] representation of the circuit, which is constructed as follows. In an ILA representation, the circuit's transition relation is converted to a CNF formula using a Tseitin transformation [18] or similar approach. Then, the formula is replicated once for each clock cycle to be modeled. Each copy of the transition relation is referred to as a *time-frame*. For each register, the next-state input from time-frame i is connected to the register output in time-frame $i + 1$. An example is shown in Fig. 1, where the circuit has been unrolled into an ILA with two time-frames.

The ILA representation of the circuit can be used to reason about the circuit's behavior over a bounded number of clock cycles. For instance, bounded model checking (BMC) [4] constructs ILA representations of the circuit with increasing numbers of time-frames, from 1 up to k. In each ILA, it asks if a particular condition can be reached in the final time-frame. If each SAT instance is unsatisfiable, that means the condition cannot occur within k clock cycles.

2.3 Property-directed reachability

While useful in verification, BMC is often only able to definitively prove that a condition cannot occur in a bounded number of clock cycles. However, it is often important to prove that a condition can never be reached in any number of clock cycles. This can be accomplished using an unbounded model checking technique, such as Property Directed Reachability (PDR) [5, 8]. For the purposes of this work, PDR can be considered as a "black box" procedure. Due to its complexity, it is not described in great detail here, only its input and output characteristics are explained. Given a circuit C, initial state set I, and a predicate

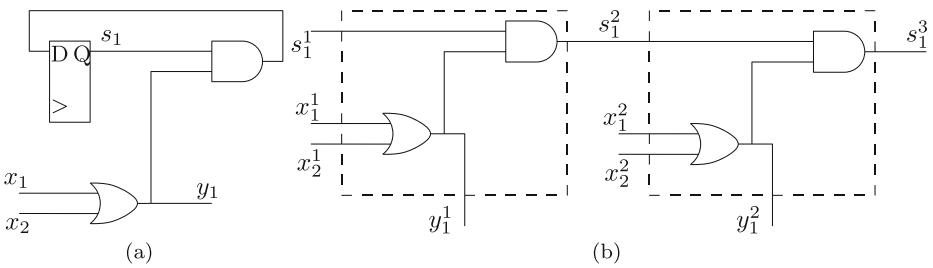

Fig. 1 **a** A sequential circuit **b** ILA representation with 2 time-frames

P representing some "unsafe" states, PDR attempts to prove that every reachable state of C is a $\neg P$-state. It either returns a certificate proving that no unsafe states are reachable or a *counter-example* trace showing that an unsafe state is reachable. In this paper, it is assumed that PDR exists as an algorithm PDR(C, I, P) that returns REACHABLE if and only if a P-state is reachable from I under circuit C. It returns UNREACHABLE otherwise.

2.4 SAT-based debugging

The work presented here draws heavily on ideas from SAT-based automated debugging [17]. Due to its importance to our work, it is explained in greater detail here. In circuit verification and debugging, a *failure* is behavior that violates the design specification. For a particular failure, an *error* is a location in the circuit that can be changed to correct the failure. Design debugging is the task of locating errors when functional verification reveals a failure.

Often, verification reveals a failure through firing assertions or observation value mismatches. In these cases, an error trace demonstrating the failure is returned. The error trace can then guide an automated debugging tool to aid the engineer in finding the error. Letting $L = \{l_1, l_2, ..., l_{|L|}\}$ denote the suspect locations, the transition relation is enhanced with a set of error select lines $e = \{e_1, e_2, ..., e_{|L|}\}$ and free variables $W = \{w_1, w_2, ..., w_{|L|}\}$. If $e_i = 0$, then the circuit's behavior is unaffected by the presence of e_i. Conversely, if $e_i = 1$, then l_i is replaced by the free variable w_i. This can be implemented by the multiplexer shown in Fig. 2.

Subsequently, the enhanced transition relation is unrolled into an ILA containing one time-frame for each clock cycle in the error trace. The error-select registers are not replicated, instead a single copy of each e_i is used across all time-frames. This ensures that the same location is replaced by its corresponding free variable across the entire ILA. Additional constraints are constructed to set the primary input values to those from the error trace and to force the primary output values to the known reference (correct) values. An additional constraint is added to force the circuit to begin in a particular initial state. Finally, cardinality constraint ϕ_n is used to ensure that exactly n error-select lines are active. The constrained ILA is converted into a CNF formula in which each satisfying assignment indicates an n-tuple of suspect locations that can be simultaneously modified to correct the erroneous behavior exposed by the error trace. Such a tuple is referred to as a *solution*. As such, an all-solutions SAT solver is used to find every solution to the problem.

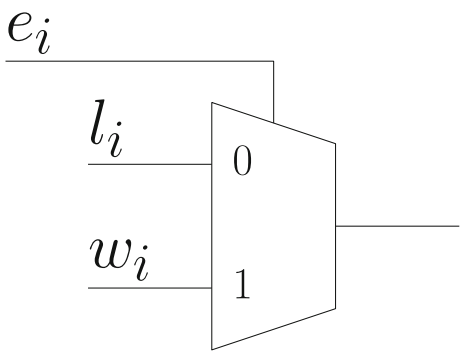

Fig. 2 Multiplexer and error-select line used in SAT-based debugging

3 Debugging unreachable states

The work presented in this paper makes extensive use of an underlying debugging algorithm. It is not tied to a particular debugging algorithm, but does require one that returns every solution in a given suspect set. That is, the underlying algorithm must return every location from its suspect set where a change can be made to make a target state reachable. To the best of our knowledge, the work of [2] is the only such algorithm. Due to its importance to our work, it is explained in greater detail here.

Given an erroneous circuit C, initial state set I, suspect set $L = \{l_1, l_2, ..., l_{|L|}\}$, and an unreachable target state predicate $\mathcal{S}$, the algorithm returns a solution set $L_{sol} \subseteq L$. The initial state set I is represented by a Boolean formula also referred to as I, since these are merely different representations of the same thing. Similarly, the target state predicate $\mathcal{S}$ is represented by a Boolean formula, and it represents the target state that is erroneously unreachable from I. Any vertex in the AIG representation of the circuit can be included in the suspect set L. The solution set L_{sol} contains every design location from L where a change can be made to make some $\mathcal{S}$-state reachable (*i.e.,* every solution in L). This additionally implies that all locations in $L \setminus L_{sol}$ are not solutions. The following example demonstrates the results of running the algorithm.

Example 1 In order to understand the characteristics the input and output of the algorithm, consider the circuit shown in Fig. 3. Assume the initial state has $s_1 = 0$ (i.e., $I = (\bar{s_1})$). The reader can verify that it is impossible to reach a state in which $s_1 = 1$. A user could choose to execute the algorithm using suspect set $L = \{l_1, l_2\}$ and target state $\mathcal{S} = (s_1)$. It returns a set of solutions $L_{sol} \subseteq L$ such that it is possible to implement a change at any location in L_{sol} so that a $\mathcal{S}$-state is reachable. In this example, the solution set is $L_{sol} = \{l_2\}$, indicating that l_2 can be modified to correct the error. Indeed, replacing the AND-gate with an OR-gate makes the target state (*i.e.,* the state where $s_1 = 1$) reachable. Other corrections are also possible. Additionally, the fact that $l_1 \notin L_{sol}$ indicates that the problem can not be fixed by changing l_1. The reader can verify that no correction is possible at l_1.

The algorithm solves a series of unbounded model checking problems using PDR. As input, PDR requires a circuit, initial state set, and unsafe state set. It returns a Boolean output that indicates if any unsafe state is reachable from any initial state. We first describe the construction of the enhanced circuit C_{en} that contains extra hardware to facilitate debugging in conjunction with PDR. Next we describe the enhanced initial state state I_{en} that appropriately constrains C_{en} to get meaningful results. Finally, it is explained how using $\mathcal{S}$ as the

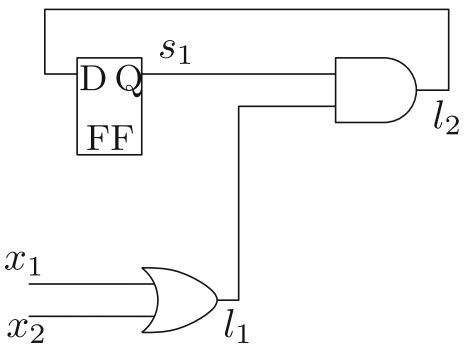

Fig. 3 Circuit in which $s_1 = 1$ is an unreachable state

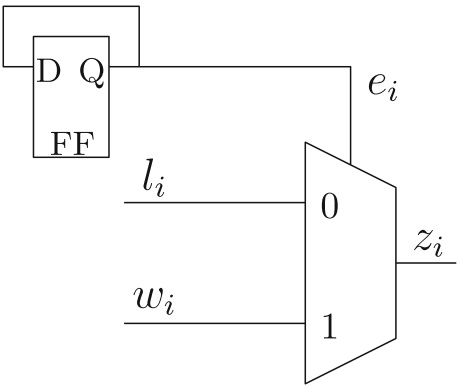

Fig. 4 Error-select register and multiplexer at suspect location l_i

unsafe state set allows the algorithm to find solutions using the results from PDR. Each step is clarified with an example.

The enhanced circuit C_{en} is constructed by the addition of extra hardware to facilitate debugging. It behaves identically to the original circuit with certain suspect locations replaced by arbitrary Boolean functions. Which locations are replaced depends on value assignments to the *error-select registers*, which are new hardware added to the original circuit. Figure 4 depicts one error-select register and the associated hardware. It can be seen that, when $e_i = 0$, the multiplexer selects the original functionality for suspect location l_i, leaving the circuit's behavior unaffected. Conversely, when $e_i = 1$, the free variable w_i is selected. Since w_i can assume any value during model checking, C_{en} behaves as though l_i is replaced by an unknown arbitrary Boolean function. The error-select register's output is fed to its input, making it immutable (i.e., its value can never change). In effect, in any trace of C_{en} the values of the error-select registers are constant. However, in different traces they can assume different values. As is explained later, this is necessary to associate the reachability of particular states with a suspect location being a solution. The following example demonstrates the construction of the enhanced circuit.

Example 2 To illustrate the behavior of C_{en}, consider the circuit depicted in Fig. 3. It has a single state element s_1, two primary inputs x_1 and x_2 and two suspect locations are labeled as l_1 and l_2. As in the previous example, assume the initial state has $s_1 = 0$, which implies that any state in which $s_1 = 1$ is unreachable. The algorithm is executed with target state $\mathcal{S} = (s_1)$. In doing so, the enhanced circuit is constructed as shown in Fig. 5. When $e_1 = e_2 = 0$, this circuit behaves the same as the original circuit. When $e_1 = 1$, l_1 is set to the free variable w_1, allowing it to assume any value during model checking. Similar behavior

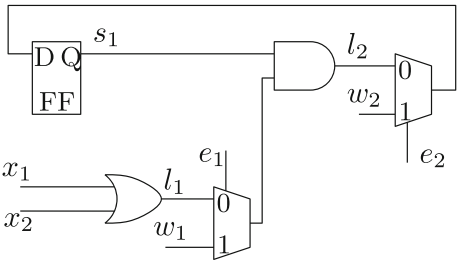

Fig. 5 Enhanced circuit

applies to e_2, l_2, and w_2. More generally, when any $e_i = 1$, the enhanced circuit behaves like the original with l_i replaced by some unknown Boolean function.

As mentioned earlier, the reachability of particular states in the enhanced circuit is associated with a specific location being a solution to the debugging problem. Consider a trace of the enhanced circuit, consisting of states $t_1, ..., t_m$. All states in the trace have the same active error-select registers by the construction of C_{en}. Let $e_1, ..., e_n$ denote the active error-select registers. The enhanced circuit therefore behaves like the original circuit with locations $l_1, ..., l_n$ replaced by arbitrary Boolean functions. It can be concluded that t_m is reachable from t_1 in the original circuit if those locations are simultaneously replaced by arbitrary Boolean functions. If $t_m \in \mathcal{S}$ and $t_1 \in I$, then a fix made at locations $l_1, ..., l_n$ can make a target state reachable. As is the case for traditional automated debugging techniques [17], the engineer is responsible for determining how to make such a fix and for re-verifying the corrected design.

For the trace to indicate a solution to the debugging problem, it must satisfy additional properties. It must start from an initial state, end on a target state, and have exactly one active error-select register e_j. The last requirement is because we assume a single error is responsible for the observed unreachability. Using the argument in the previous paragraph, replacing l_j with a different Boolean function makes a target state reachable. This implies l_j is a solution. Solutions can be found by finding traces that satisfy these three properties. The approach can also be extended to handle higher error cardinalities n by requiring exactly n active error-select registers. In that context, a solution is an n-tuple of suspect locations. However, for the purposes of this paper, solutions are restricted to single locations (i.e., an error cardinality of one).

This motivates the construction of the enhanced initial state set I_{en}. The original registers of the circuit are constrained with I. This forces C_{en} to start on an initial state of the original circuit. Since exactly one error-select register must be active, a cardinality constraint [17] is used to constrain the error-select registers. The cardinality constraint ϕ ensures that exactly one error-select register is active. The enhanced initial state set is therefore represented by the formula $I_{en} = I \wedge \phi$. The following example clarifies this process.

Example 3 Consider again the example from Fig. 3. The enhanced initial state condition I_{en} is the conjunction of $I = (\bar{s_1})$ and the cardinality constraint ϕ. Therefore, $I_{en} = (\bar{s_1}) \wedge (e_1 \vee e_2) \wedge (\bar{e_1} \vee \bar{e_2})$. The set of states in I_{en} is $\{(\bar{s_1} \wedge e_1 \wedge \bar{e_2}), (\bar{s_1} \wedge \bar{e_1} \wedge e_2)\}$. Notice that these are all states in which $s_1 = 0$, which is the initial state condition. Additionally, every state of I_{en} has one active error-select register, matching the requirements for traces that indicate solutions.

All that remains is to provide the model checker's unsafe state set. As mentioned earlier, three properties are needed for a trace to indicate a solution. The enhanced circuit and initial state set ensure that any trace found begins at an initial state and has exactly one active error-select register. The only remaining property is that the trace must end on a target state. Therefore, using $\mathcal{S}$ as the unsafe state set ensures all three properties are met. The following example demonstrates how a solution is found using the model checker.

Example 4 Continuing the illustration of the methodology from the previous example, recall that the target state condition is $\mathcal{S} = (s_1)$ and the initial state condition is $I = (\bar{s_1})$. The enhanced circuit has the following counter-example trace: $\langle t_0, t_1 \rangle = \langle (\bar{s_1} \wedge \bar{e_1} \wedge e_2), (s_1 \wedge \bar{e_1} \wedge e_2) \rangle$. Notice that t_0 corresponds to an initial state of the original circuit, t_1 is a target

state, and e_2 is the active error-select register. In states t_0 and t_1 the model behaves identically to the original circuit with l_2 replaced by an unknown function. Since t_0 is an initial state and t_1 is a target state, replacing l_2 with a different function makes a target state reachable in the original circuit. This indicates that location l_2 is a solution. Indeed, the reader can confirm that replacing the AND-gate that drives l_2 with an OR-gate makes the target state reachable. Other corrections to the problem are also possible.

When a solution is found, I_{en} is updated to force the corresponding error-select register to `0` so that the algorithm can search for additional solutions. For a solution l_j, this is done by conjoining the unit literal clause $\neg e_j$ to the formula for I_{en}. Doing so prevents PDR from finding additional counter-examples in which e_j is active. The model checker is called with the updated version of I_{en}. This process continues until the model checker indicates that $\mathcal{S}$ is unreachable. This implies that no single location in $L \setminus L_{sol}$ can be changed to make a target state reachable. In other words, L_{sol} contains every solution from L and the algorithm can terminate. The following example demonstrates the process of finding and blocking a solution.

Example 5 For the circuit of Fig. 3, after finding the solution l_2, the enhanced initial state condition becomes $I_{en} = (\bar{s_1}) \wedge (e_1 \vee e_2) \wedge (\bar{e_1} \vee \bar{e_2}) \wedge (\bar{e_2})$, leaving $(\bar{s_1} \wedge e_1 \wedge \bar{e_2})$ as the only remaining initial state. It is easily verified that this state cannot reach any target states, implying that l_1 is not a solution. This is indeed the case. To reach a state where $s_1 = 1$ the output of the AND-gate must be `1`. In the initial state $s_1 = 0$, so regardless of the value at l_1 the AND-gate will never output `1`. Therefore, there is no way to modify the circuit at l_1 to rectify the unreachability of the target state.

The steps of the algorithm are shown in Algorithm 1. Lines 1 through 3 perform initialization. Lines 4 through 8 contain the main loop in which solutions are found. Line 4 runs PDR to either find a solution or conclude that no more solutions exist. Line 5 extracts the solution from the trace returned by PDR. The following line updates the solution set, and then line 7 blocks the solution from appearing in future traces. Finally, line 9 returns the solution set.

The approach is both sound and complete with respect to its input suspect set. That is, every location in L_{sol} is a solution, and L_{sol} contains every solution from L. Soundness follows directly from the construction of C_{en} – if a trace reaches $\mathcal{S}$ then it has exactly one error-select register assigned to `1` and that error-select register corresponds to a solution. Completeness follows from the completeness of the underlying unbounded model checking technique of PDR. Note that, while not described here, this approach can easily be extended to handle higher error cardinalities. That is, it can be extended to find solutions that consist of an n-tuple of suspect locations, where all n must be modified simultaneously to correct the error.

4 Efficient suspect selection

This section presents an algorithm that uses Algorithm 1 to find the complete solution set without choosing L to include every design location. Using this methodology, only a subset of the design locations are passed to the underlying debugging algorithm. Despite that, the methodology still returns the complete solution set to the problem. However, this depends on the assumption that only a single design error is responsible for the observed failure, or

Algorithm 1 UNREACHABILITY($C, I, \mathcal{S}, L$)

1: $L_{sol} = \emptyset$
2: Construct C_{en}
3: $I_{en} = I \wedge \phi$
4: **while** $PDR(C_{en}, I_{en}, \mathcal{S})$ == REACHABLE **do**
5: $\quad e_j =$ active error-select register in counter-example
6: $\quad L_{sol} = L_{sol} \cup \{l_j\}$
7: $\quad I_{en} = I_{en} \wedge (\neg e_j)$
8: **end while**
9: **return** L_{sol}

in other words, the error cardinality is one. In practice, this is commonly the case, meaning that this is often not a significant limitation. It may be possible to extend the approach to handle higher error cardinalities, but doing so is expected to require examining many more suspect locations, reducing the performance benefits gained by this approach.

4.1 Suspect selection methodology

The algorithm accepts as input a circuit C, initial state set I, and predicate $\mathcal{S}$ representing the unreachable target state. As was the case for Algorithm 1, I and $\mathcal{S}$ are defined by Boolean formulas over the state elements of C. Unlike that algorithm, the user does not specify a suspect set. Instead, the approach completes a series of iterations, each of which calls Algorithm 1 with a different suspect set. The suspect set in each iteration is constructed from the solution set obtained in the previous iteration.

Toward this end, the algorithm begins with a preprocessing step on the AIG representation of the circuit. The set of all *fanout points* is computed. A fanout point is simply a vertex with an out-degree greater than one. Figure 6 depicts this concept graphically for both a circuit and its AIG representation. Let $F = \{l : |fanout(l)| > 1\}$ denote the set of all fanout points. Additionally, let R denote the set of all registers that appear in the formula representing $\mathcal{S}$. These sets are used to construct the suspect set for the first iteration. The rationale behind this step is explained in greater detail later in this section.

After preprocessing, the algorithm proceeds through a series of iterations, each of which makes one call to Algorithm 1. Each iteration uses a different suspect set, constructed to limit the total number of suspects examined across all iterations. Let L_i (resp. S_i) denote the suspect set (resp. solution set) for iteration i. The initial suspect set is constructed as $L_1 = R \cup F$, so it includes all fanout points and all registers that appear in the target state

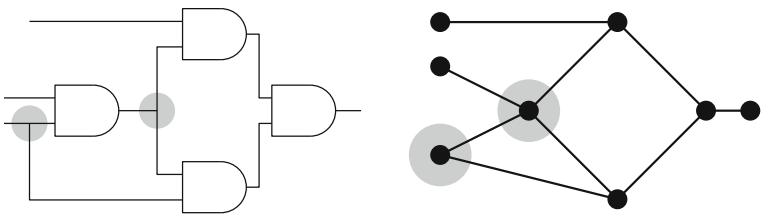

Fig. 6 **a** Example circuit with fanout points highlighted **b** Equivalent AIG

predicate. Algorithm 1 is then executed using this suspect set, returning a set of solutions S_1. Upon completion of iteration i, a new suspect set L_{i+1} is computed as shown in (1) below.

$$L_{i+1} = \{l \in S_i : fanin(l)\} \setminus \bigcup_{j=1}^{i} (L_j) \tag{1}$$

Note that L_{i+1} contains the fanin of every solution in iteration i. However, it excludes any suspects that were used in previous iterations. Otherwise, the algorithm might consider the same suspect more than once. This ensures that no location is a suspect in multiple iterations, guaranteeing that the algorithm makes progress in every iteration and therefore terminates.

The reasoning behind this approach is intuitive. If a location l is a solution, then l can be replaced by a different Boolean function to make a target state reachable. It also may be possible to make the change at a fanin of l, so the algorithm must also check if the fanin locations are solutions. This occurs if the change needed at l is equivalent to changing only one of its fanin locations. Alternatively, it may not be possible to correct the error at any fanin of l, and therefore l is a solution but no element of $fanin(l)$ is. For example, in the figure in Fig. 7, it may not be possible to fix the circuit by making a change at the highlighted location, but not by making a change at either of the OR-gates in its fanout. As a result, the fact that l is a solution is not sufficient to prove whether or not its fanin locations are solutions. They are therefore included in a suspect set.

Conversely, consider a location l' that is not a solution. There is no way to modify the design at l' to fix the error. In some cases, this may imply that the locations in the fanin of l' are not solutions. Consider a location $l \in fanin(l')$. If l has other fanout locations besides l', it may be possible for l to be a solution even if all of its fanout locations are not. This case can occur if multiple fanouts of l need to be corrected to fix the error. Similarly, if $l \in R$, then it may be the case that l is a solution but none of its fanout locations are. However, if $|fanout(l)| = 1, l \notin R$, and the single fanout of l is not a solution, then l is not a solution. The following lemma formalizes this notion.

Lemma 1 *For a location $l \notin (R \cup F)$, if the single element of $fanout(l)$ is not a solution, then l is not a solution.*

Proof Since $l \notin F$, we have $|fanout(l)| = 1$. Suppose that l is a solution and that the single element $l' \in fanout(l)$ is not a solution. This implies that it is possible to replace l by some other Boolean function to make some $\mathcal{S}$-state reachable. Also, since $l \notin R$ but l is a solution, l is in the COI of R and either $l' \in R$ or l' is in the COI of R. Otherwise, a change at l would not be observable at R and could not correct the error.

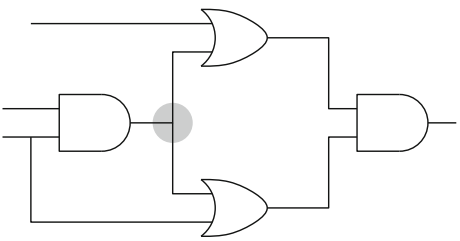

Fig. 7 Example circuit with fanout

Algorithm 2 UNREACHABILITYENHANCED($C, I, \mathcal{S}$)

1: $R =$ state elements in the formula defining $\mathcal{S}$
2: $F = \{l : |fanout(l)| > 1\}$
3: $L_1 = F \cup R$
4: $i = 1$
5: **while** $S_i =$UNREACHABILITY$(C, I, \mathcal{S}, L_i) \neq \emptyset$ **do**
6: $\quad L_{i+1} = \{l \in S_i : fanin(l)\} \setminus \bigcup_{j=1}^{i}(L_i)$
7: $\quad i = i + 1$
8: **end while**
9: $S = \bigcup_{j=1}^{i} S_j$
10: **return** S

However, since l' is not a solution, there is no way to replace l' with a different Boolean function to make an $\mathcal{S}$-state reachable. Since l' is the only fanout of l, this implies that it is possible to replace l in a manner that changes the behavior at R but not at l'. This is clearly a contradiction, since the behavior of the circuit must also change at l' to be observable at R. □

Note that $l \notin (R \cup F)$ can be restated as $l \notin L_1$. This is the rationale behind the construction of L_1. It includes every location that is not handled by Lemma 1. Therefore, the algorithm never needs to check a location that is only in the fanin of non-solution locations. The next subsection presents theorems based on this fact demonstrating the soundness and completeness of the approach.

The steps of the approach are shown in Algorithm 2. In that description, algorithm UNREACHABILITY is Algorithm 1. Lines 1 and 2 construct R and F, respectively. Line 3 constructs the initial suspect set. Lines 5–8 contain the main loop that repeatedly calls UNREACHABILITY. Within the loop, the suspect set for the next iteration is constructed on line 6 according to (1). Line 9 constructs the solution set from the solution sets of each iteration. Finally, line 10 returns the solutions.

4.2 Soundness and completeness

The previous subsection describes the approach and the intuitive rationale behind the manner in which suspect sets are constructed. This subsection proves that the reasoning is correct and that the approach is both sound and complete. In this context, soundness implies that every location Algorithm 2 returns is indeed a solution. Completeness requires that it also returns every solution in the circuit. The soundness of the algorithm, stated in Theorem 1, follows immediately from the soundness of Algorithm 1.

Theorem 1 *Every location in S is a solution*

Proof Immediate from soundness of Algorithm 1 □

Since S is the solution set of Algorithm 2, this proves that the algorithm is sound. Theorem 2 below uses the construction of L_1 along with Lemma 1 to demonstrate that the algorithm is complete.

Theorem 2 *When Algorithm 2 terminates, S includes every solution.*

Proof The initial suspect set is $F \cup R$. Since Algorithm 1 is complete, S includes all solutions from $F \cup R$. For every location l not in the initial suspect set, $l \notin R$ and $|fanout(l)| \leq 1$. If $|fanout(l)| = 0$, l is not in the COI of any location other than itself. Therefore, if $l \notin R$ and $|fanout(l)| = 0$, l is not a solution as it is clearly not in the COI of R. Therefore, by Lemma 1, these locations are only solutions if they are in the fanin of other solutions. On Line 6 of Algorithm 2, a new suspect set is constructed including the fanin of all solutions found in the previous iteration. It continues in this manner until it reaches an iteration in which no solutions are found. As a result, any location in the fanin of any solution is included in a suspect set passed to Algorithm 1. Therefore, by the completeness of Algorithm 1, all of these solutions are found as well and S contains every solution when the algorithm terminates. □

Since S is the solution set of Algorithm 2, Theorem 2 proves that the algorithm is complete. In contrast with Algorithm 1, the algorithm does not require the user to specify a set of suspect locations. Since the algorithm intelligently selects the suspect sets it uses, it essentially performs this step for the user. However, it is possible for the user to have additional knowledge regarding the source of the error. For instance, if the user were to introduce a bug when modifying a specific module, it may be beneficial to restrict the suspect set to locations within that module in order to improve the algorithm's runtime. While Algorithm 2 does not provide this functionality, it can be easily achieved by adding a set of trusted locations that are never allowed to be included in the suspect set constructed on line 6. Empirical results presented in the next section demonstrate that this is not necessary in most cases, as the algorithm only considers a relatively small portion of the circuit as suspects.

5 Experimental results

All results presented in this section are run on a single core of an i5-3570K 3.4-GHz workstation with 16 GB of RAM. The presented algorithm is built on top of an implementation of Algorithm 1 [2] using a reference implementation of PDR [5]. Two sets of designs are considered as benchmarks. The first consists of six designs from OpenCores [15] and one commercial design of a serial interface from an industrial partner. Each problem instance is created by injecting a design error that makes a state erroneously unreachable. Examples of design errors include incorrect operators in expressions, complemented conditions in if-statements, added incorrect state transitions, etc. These are all typical design errors observed in industry. The second set of benchmarks consists of 17 circuits from the Hardware Model Checking Competition (HWMCC) [7], and represents very hard problem instances. The proposed algorithm is compared against Algorithm 1, where the suspect set L is chosen to include every location in the circuit.

Table 1 shows comprehensive results. The first 17 rows relate to the HWMCC circuits, and the following 7 rows relate to the OpenCores and commercial designs. The first column shows the name of the problem instance, The next three columns show the size of the suspect set, number of solutions found, and runtime using Algorithm 1 as described above. The remaining five columns relate to Algorithm 2. They show the number of solutions found, number of iterations executed, total number of suspects considered across all iterations, runtime, total percentage of suspects considered ($|\bigcup L_i|/|L|$), and speedup relative to Algorithm 1, respectively.

It can be seen that both algorithms always find the same set of solutions, as expected. Across all experiments, Algorithm 2 offers a geometric mean speedup of 15× with a median

Table 1 Experimental results

	Algorithm 1			Algorithm 2					
bench-mark	$\|L\|$	#sol	time (s)	#sol	#it	$\|\bigcup L_i\|$	time (s)	% sus	spee-dup
beemlup1b1	2634	144	372.9	144	86	836	8.0	31.74%	46.4×
bjrb07amba1	1051	40	172.7	40	5	203	7.3	19.31%	23.8×
cmugigamax	661	299	15.8	299	28	168	3.0	25.42%	5.2×
kenflashp01	1345	386	13.8	386	79	309	1.6	22.97%	8.4×
kenflashp11	6366	155	415.7	155	47	956	12.6	15.02%	33.1×
kenoopp1	666	85	2.2	85	43	204	0.5	30.63%	4.3×
pdtvisgigamax3	1087	333	205.8	333	10	246	28.1	22.63%	7.3×
pdtvistwoall3	1752	240	30.3	240	8	325	4.6	18.55%	6.6×
pdtvisvending09	985	22	8.2	22	3	144	0.5	14.62%	16.7×
pdtvsarmultip13	2873	76	771.8	76	5	294	9.1	10.23%	85.1×
power2bit128	113	84	200.8	84	6	20	42.7	17.70%	4.7×
power2sum128	240	215	786.1	215	17	38	212.9	15.83%	3.7×
power2sum256	246	221	1783.3	221	17	38	513.3	15.45%	3.5×
power2sum32	228	203	109.8	203	17	38	25.0	16.67%	4.4×
shift1add512	119	59	559.9	59	8	32	80.8	26.89%	6.9×
vis4arbitp1	337	144	305.5	144	7	60	15.2	17.80%	20.2×
viselevatorp3	1173	173	69.5	173	13	256	3.6	21.82%	19.2×
mrisc_core	9573	18	276.9	18	6	1696	6.1	17.72%	45.6×
design1	1208	9	11.4	9	4	227	0.4	18.79%	25.9×
divider	3915	38	419.6	38	3	1021	12.2	26.08%	34.3×
spi	1156	23	7.9	23	7	230	0.7	19.90%	11.6×
wb	451	193	10	193	8	52	0.5	11.53%	19.7×
usb_core	5545	6	631.4	6	3	1137	3.5	20.50%	194.3×
ac97_ctrl	14967	13	496.4	13	4	2688	17.9	17.96%	27.8×
GEOMEAN									15.0×
MEDIAN								18.7%	17.9×

of 17.9×. The proposed approach is able to safely ignore a majority of all design locations. Across all experiments, the proposed approach is able to ignore a median of 81.3% of all design locations, or equivalently, it considers 18.7% of design locations. Since the runtime for the previous approach appears to be heavily influenced by the size of the suspect set, eliminating the majority of locations from consideration seems to yield a substantial reduction in runtime.

Figure 8 plots the number of solutions and non-solution suspects for each iteration for the `spi` problem instance. It can be seen that the suspect set of the first iteration is drastically larger than that of subsequent iterations. This occurs because the majority of locations are not solutions. Many suspects are considered in the first iteration, and only a small portion of them are found to be solutions. In the subsequent iterations only a subset of the locations in the fanin of previously found solutions can be part of the suspect set. In most cases this represents a very small portion of the design. In the case of `spi`, it can be seen that the first iteration uses a suspect set with 229 locations, only 7 of which are found to be solutions. In

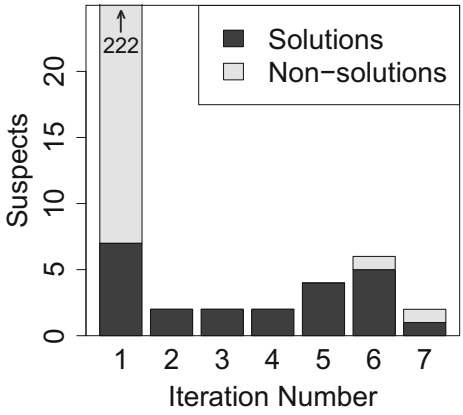

Fig. 8 Solutions and non-solutions per iteration for `spi`

the following iteration, only locations in the fanin of these 7 solutions can be considered, giving a much smaller suspect set.

Table 2 shows the number of solutions and number of suspects per iteration for selected benchmarks. It demonstrates that in many cases, the initial suspect set contains few solutions, meaning that very few suspects are considered in subsequent iterations. The `wb` problem instance is one exception. This occurs because a fairly large portion of the design locations are solutions. Even so, the algorithm appears to be highly efficient at ignoring non-solutions locations, as it only considers a total of 237 suspect locations in order to find 193 solutions. Even in this somewhat pathological case, the proposed algorithm is able to ignore nearly half of the design locations and achieve a 19× speedup over the previous approach.

The runtime of the the previous approach appears to be heavily-dependent on the suspect set it is given. It can be seen in Fig. 9, which plots the runtime of each iteration for `spi`, that the first iteration consumes substantially more runtime than later iterations. This appears to confirm that larger suspect sets require more runtime to solve. This is not surprising, as a larger suspect set substantially increases the complexity of the PDR instances solved by UNREACHABILITY. It is additionally expected that suspect sets with many non-solutions impact the algorithm's runtime more substantially than those with many solutions. To find a solution, PDR simply needs to find a counter-example trace that reaches a target state. Conversely, to prove locations are not solutions PDR must prove that no such counter-example exists. This seems to be an inherently difficult problem. When a large number of

Table 2 Suspects L_i and solutions for selected benchmarks S_i in each iteration of Algorithm 2

benchmark	$\lvert S_1\rvert/\lvert L_1\rvert$	$\lvert S_2\rvert/\lvert L_2\rvert$	$\lvert S_3\rvert/\lvert L_3\rvert$	$\lvert S_4\rvert/\lvert L_4\rvert$	$\lvert S_5\rvert/\lvert L_5\rvert$	$\lvert S_6\rvert/\lvert L_6\rvert$
bjrbamba1	14/202	10/11	10/15	4/5	2/3	0/3
pdrvsarmultip13	23/297	20/20	26/35	5/11	2/2	–
power2bit128	23/28	17/18	19/22	9/9	8/8	8/12
mrisc_core	4/1688	4/4	2/2	3/4	3/6	2/4
divider	10/1028	10/10	18/18	–	–	–
spi	7/229	2/2	2/2	2/2	4/4	5/6
wb	33/76	33/34	33/33	34/34	4/4	8/8
ac97_ctrl	5/2689	2/2	2/2	2/2	2/2	–

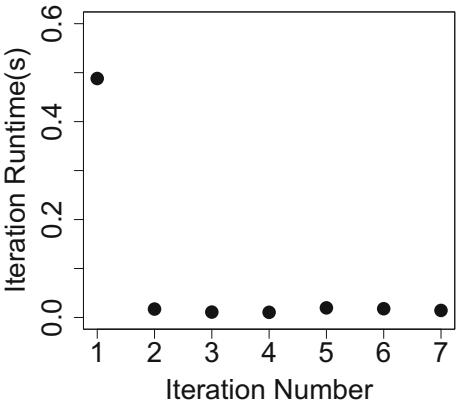

Fig. 9 Runtime for each iteration for `spi`

non-solution locations are in the suspect set, proving no counter-examples exist can be an expensive operation due to increased complexity of the model used in PDR.

Figure 10 confirms this intuition. It plots the number of solutions found over time for `spi` for both approaches. It can be seen that the previous approach appears to find many solutions toward the beginning of the run. These solutions result from counter-examples that PDR is able to find more easily. After exhausting the easy counter-examples, it begins to take longer to find later solutions. Finally, after finding all solutions, the algorithm also takes a substantial amount of time to prove no further solutions exist before terminating.

Conversely, Fig. 10b shows that Algorithm 2 finds few of its solutions at the start of the run. This is because the first iteration has a large suspect set. It can be seen that after finding 7 solutions (all of the solutions for iteration 1), there is a substantial gap before finding the eighth solution. This gap represents the time required to prove that the non-solution locations in the set L_1 are in fact not solutions. As L_1 is a relatively large suspect set, this takes a significant amount of time. After the conclusion of iteration 1, the suspect sets are all much smaller than L_1. As a result, each iteration requires very little runtime and many solutions are found in a short period of time. This confirms that using Lemma 1 to limit the suspect sets is a highly effective means of accelerating the debugging process.

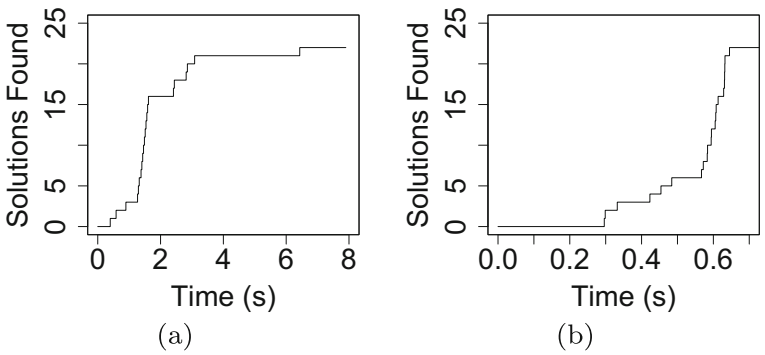

Fig. 10 Solutions found vs. running time (`spi` benchmark) for **a** the previous approach **b** Algorithm 2

6 Conclusion

This work presented an algorithm to diagnose errors that cause unreachable states. The presented algorithm returns the complete solution set of the problem without exhaustively examining every design location. It improves upon the previous technique by ignoring a large portion of the design locations that are provably not solutions. Empirical results confirmed that a majority of design locations are safely ignored resulting in a substantial runtime improvement relative to the previous approach.

References

1. Berryhill, R., Veneris, A.: Automated Rectification Methodologies to Functional State-Space Unreachability. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pp. 1401–1406 (2015)
2. Berryhill, R., Veneris, A.: A Complete Approach to Unreachable State Diagnosability via Property Directed Reachability. In: Proceedings of the 2016 Asia and South Pacific Design Automation Conference, ASP-DAC '16 (2016)
3. Berryhill, R., Veneris, A.: Efficient Selection of Suspect Sets in Unreachable State Diagnosis. In: Proceedings of the 2016 Int'l Symposium on Artificial Intelligence and Mathematics, ISAIM '16 (2016)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. In: Advances in Computers, vol. 58, pp. 118–149 (2003)
5. Bradley, A.: Sat-Based Model Checking without Unrolling. In: Int'l Conf. on Verification, Model Checking, and Abstract Interpretation, pp. 70–87 (2011)
6. Brummayer, R., Biere, A.: Local Two-Level And-Inverter Graph Minimization without Blowup. In: Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS '06 (2006)
7. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K., Baumgartner, J.: Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks vol. 9 (2016)
8. Eén, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, pp. 125–134. FMCAD Inc, Austin (2011)
9. Foster, H.: Assertion-Based Verification: Industry Myths to Realities (Invited Tutorial). In: Int'l Conference on Computer-Aided Verification (CAV), pp. 5–10 (2008)
10. Foster, H.: From Volume to Velocity: The Transforming Landscape in Function Verification. In: Design Verification Conference (2011)
11. Keng, B., Safarpour, S., Veneris, A.: Bounded model debugging. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **29**(11), 1790–1803 (2010). https://doi.org/10.1109/TCAD.2010.2061370
12. Keng, B., Veneris, A.: Path-directed abstraction and refinement for sat-based design debugging. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(10), 1609–1622 (2013). https://doi.org/10.1109/TCAD.2013.2263036
13. Mangassarian, H., Veneris, A., Safarpour, S., Benedetti, M., Smith, D.: A Performance-Driven Qbf-Based on Iterative Logic Array Representation with Applications to Verification, Debug and Test. In: Int'l Conf. on CAD (2007)
14. Mangassarian, H., Le, B., Veneris, A.: Debugging rtl using structural dominance. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(1), 153–166 (2014). https://doi.org/10.1109/TCAD.2013.2278491
15. OpenCores.org: http://www.opencores.org (2007)
16. Safarpour, S., Veneris, A.: Automated design debugging with abstraction and refinement. Trans. Comp. Aided Des. Integ. Cir. Sys. **28**(10), 1597–1608 (2009)
17. Smith, A., Veneris, A., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using boolean satisfiability. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **24**(10), 1606–1621 (2005)
18. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic Part II, 115–125 (1968)