# Efficient Selection of Suspect Sets in Unreachable State Diagnosis

**Ryan Berryhill** and **Andreas Veneris**
University of Toronto, Department of Electrical and Computer Engineering
10 King's College Road
Toronto, Ontario M5S 3G4

**Abstract**

In the modern hardware design cycle, correcting the design when functional verification reveals an erroneously unreachable state can be a time-consuming manual process. In order to mitigate this growing cost, this paper presents an automated methodology that, given an unreachable target state, returns all design locations where a change can be implemented to make it reachable. In contrast to previous automated unreachability debugging techniques, our approach avoids exhaustively examining all design locations, resulting in significant run-time savings. The presented approach proceeds through a series of iterations, each of which examines a subset of the design locations (suspects) to determine whether or not they are solutions to the problem. Based on the results of each iteration, a new set of suspects is considered in the subsequent iteration. However, in practice a small portion of the locations in the design are ever examined. Results are presented to prove that when the initial suspect set is chosen appropriately, the solution set returned by the methodology is complete. Empirical results on industrial designs confirm the theoretical and practical gains of this approach, demonstrating an impressive 33.7x speedup over the previous approach while avoiding examining 76% of the design locations on average.

## 1 Introduction

Functional verification has grown to become the primary bottleneck in the modern hardware design cycle, accounting for up to 70% of the total design effort (Foster 2011). Within the verification cycle, debugging accounts for up to half of the time consumed (Foster 2008). A great deal of automation is available to aid engineers in the verification and debugging process, somewhat mitigating the substantial resources they require. While invaluable to the debugging process, some of these techniques can be difficult to apply in practice, particularly when considering large designs which may require substantial computational resources to verify and debug.

When functional verification reveals an error such as an observation value mismatch, a scoreboard discrepancy, or a firing assertion, an error-trace is returned that demonstrates the problem. A SAT-based automated debugging utility (Smith et al. 2005; Huang and Cheng 1998; Chang, Markov, and Bertacco 2007; Fey et al. 2008) can be then applied to aid the engineer in finding the root cause of the error so it can be fixed. Functional verification may also reveal that a state is unreachable in violation of the design specification. In this case an error is detected, but no error-trace is available to guide traditional automated debugging utilities. Specialized SAT-based automated debugging techniques (Berryhill and Veneris 2015a; 2015b; 2016) exist to find the root cause of these errors. The approach of (Berryhill and Veneris 2015a) involves a series of intertwined iterations of reachable state space approximation and traditional SAT-based debugging. While generally effective it is not complete by nature, and may only return a subset of the solutions to the problem. Conversely, the work in (Berryhill and Veneris 2016) provides an approach that is complete with respect to its input set of suspect locations. That is, given a set of suspect locations, it is guaranteed to return all locations where a change can be made to correct the error (*i.e.,* solutions) that are in that set. However, its runtime performance may degrade substantially when the suspect set is large, limiting its applicability in debugging large designs with numerous potential suspect locations.

Towards the goal of alleviating this problem, this paper presents a novel automated debugging methodology based on the work of (Berryhill and Veneris 2016) that obviates the need to specify a set of suspect locations. Given an unreachable target state, the methodology returns every location in the circuit where a change can be made to make the target state reachable. In contrast with the previous approach, it does not require the user to specify a set of suspect locations and its runtime does not appear to explode with larger suspect sets. The methodology proceeds through a series of iterations, where each iteration executes the previous approach with a set of suspects that is small relative to the size of the circuit. The key innovation is that the non-solution suspects may indicate other locations that are also not solutions and therefore can be safely ignored in future iterations. In this manner, the algorithm is able to safely ignore many suspect locations and thereby avoid the runtime explosion characteristic to the previous approach. Experimental results show that the algorithm is able to discard 76% of the circuit's locations as non-solutions without providing them to the underlying algorithm as suspects.

In greater detail, the algorithm works as follows. The only input it requires is a predicate representing the target state. First, the initial suspect set is constructed to include the registers that appear in the target state predicate and all locations in the circuit with fanout. The debugging algorithm is then executed with this suspect set. When a solution is found, it is recorded to later be returned to the user. Additionally, all of its fanins except for those that have already been in the suspect set of a previous iteration are added to the suspect set of the subsequent iteration. The algorithm continues executing new iterations until either no locations are left or an iteration runs and finds no solutions. In this manner, any location that only appears in the fanin of known non-solution locations is never included in a suspect set. In practice, since the solution set of the problem tends to be small, most locations are ignored and the size of the suspect set at each iteration is small. However, as all ignored locations are known not to be solutions, the algorithm is complete.

Experiments on industrial designs demonstrate the theoretical findings and the effectiveness of the proposed approach. In all cases, the proposed methodology finds the same solution set as the previous approach, while providing a geometric mean 33.7x speed-up. Additionally, it is found that the average number of iterations is 5.1 and that an impressive 76% of the circuit's locations are safely ignored.

The remainder of this paper is organized as follows. Section 2 presents background material regarding existing techniques for debugging unreachable states. Section 3 presents the proposed technique along with proofs of its soundness and completeness. Section 4 presents empirical results that confirm the effectiveness of the proposed technique. Finally section 5 concludes the paper.

## 2 Preliminaries

### 2.1 Notation

The following notation is used throughout this paper. Each assignment to the state elements of a sequential circuit $C$ represents 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 input that causes $C$ to transition from $t$ to $t'$. 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 referred in this paper as 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) (Brummayer and Biere 2006). In an AIG, the circuit is represented by 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 (latch). An AND-gate vertex has two in-edges representing the inputs to the gate and one or more out-edges representing its output. Similarly, a NOT-gate vertex has a single in-edge representing the 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 latch is represented by two vertices: its next-state input with has in-degree one and out-degree zero, and its output with out-degree of at least one.

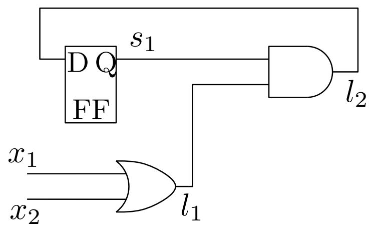

Figure 1: Circuit in which $s_1 = 1$ is an unreachable state

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, this includes any location $l'$ for which an edge $(l', l)$ exists. Additionally, if $l$ is the output of a latch, $fanin(l)$ is the next-state input of the latch. 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 1-step cone-of-influence (COI) for a location $l$ is the set of all locations $l'$ for which the AIG contains a path from $l'$ to $l$. The $i$-step cone-of-influence for a location $l$ is the set of all locations $l'$ for which a path exists from $l'$ to an element of the $(i-1)$-step cone-of-influence. The cone-of-influence for $l$ is the set of locations that appear in any $i$-step COI of $l$.

### 2.2 Prior Work

The work presented here extensively utilizes the automated debugging approach of (Berryhill and Veneris 2016). Given an erroneous circuit $C$, a set of suspect locations $L$ (the *suspect set*), and an unreachable target state predicate $\mathcal{S}$, the approach finds locations from $L$ where a change can be implemented to make some $\mathcal{S}$-state(s) reachable. Any such location is a *solution*. Any vertex of the circuit's AIG representation may be considered a suspect. The algorithm is complete with respect to its input set. In other words, it returns a solution set $L_{sol} \subseteq L$ where all solutions from $L$ are included in $L_{sol}$. The algorithm is also sound, meaning that all locations in $L_{sol}$ are solutions. These facts also imply that when the algorithm terminates, all locations in $L \setminus L_{sol}$ are also proven not to be solutions.

For example, consider the circuit of Figure 1, and assume the initial state has $s_1 = 0$. One can observe that it is impossible for the circuit to reach a state in which $s_1 = 1$. The algorithm can be executed with $\mathcal{S} = (s_1)$ as a target state and $L = \{l_1, l_2\}$ as its suspect set. The algorithm 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 state in which the predicate $\mathcal{S}$ is true is made reachable. In this example, the solution set is $L_{sol} = \{l_2\}$ indicating that $l_2$ can be modified to correct the error. Indeed, it is possible to replace the AND-gate with an OR-gate to make the target state reachable. Other corrections are also possible. The fact that $l_1 \notin L_{sol}$ indicates that no correction is possible at $l_1$. This is easily verified as in the initial state $s_1 = 0$, so no matter what the output of $l_1$ is, the AND-gate will never output a 1 and the circuit will never enter the target state.

The algorithm works by constructing an enhanced model of the circuit with added hardware to facilitate debugging. The model is constructed such that any $\mathcal{S}$-state is reachable under the model if and only if one or more of the suspect locations is a solution. The algorithm uses the unbounded model checking technique of Property-Directed Reachability (PDR) (Bradley 2011) to determine if any $\mathcal{S}$-state is reachable. PDR is a model checking technique that solves multiple SAT instances in order to either prove that $\mathcal{S}$ is unreachable or compute a counter-example trace that causes the circuit to reach an $\mathcal{S}$-state.

If an $\mathcal{S}$-state is reachable, the answer from PDR will also indicate a suspect location that is a solution. Subsequently, this location is removed from $L$ and the process is repeated until no more solutions exist. In this manner, the algorithm finds all solutions to the problem.

In this paper, it is assumed this approach exists as the algorithm UNREACHABILITY($C, L, \mathcal{S}$). The algorithm accepts a circuit $C$, set of suspect locations $L$, and an unreachable target state condition $\mathcal{S}$, and it returns $L_{sol}$ as described above.

## 3 Selecting Suspects Efficiently

This section presents a technique based on the work of (Berryhill and Veneris 2016) that localizes bugs that cause unreachable states. Given an erroneous circuit $C$ and an unreachable target state condition $\mathcal{S}$, the proposed technique finds all locations in the circuit where a change can be implemented to make an $\mathcal{S}$-state reachable. The target state condition $\mathcal{S}$ is a predicate defined by a Boolean formula over the state elements of the circuit and is specified by the user.

### 3.1 Methodology

The algorithm involves a series of iterations of the previous technique in which each iteration's suspect set is chosen to limit the number of suspects considered across all iterations. In a given iteration, each solution found is used to add suspect locations to the suspect set of the subsequent iteration. Conversely, a location proven not to be a solution may indicate that other locations are also not solutions and can safely be ignored.

Towards this end, the algorithm begins with a preprocessing step. The circuit is converted into its AIG representation and the set of all *fanout points* in the AIG is computed. A fanout point is simply a vertex with an out-degree greater than 1. Figure 2 depicts this concept graphically for both a circuit and its equivalent AIG. It can be seen that a fanout point is simply a location that fans out to the inputs of more than one other vertex (latch, gate, or output). Let $F = \{l : |fanout(l)| > 1\}$ denote the set of all fanout points. Additionally, the set $R$ of all latches that appear in the target state predicate $\mathcal{S}$ is computed. The exact rationale and importance of these sets is explained later in this section.

After preprocessing, the algorithm proceeds through a series of iterations, each of which makes one call to the algorithm UNREACHABILITY. Each iteration uses a different

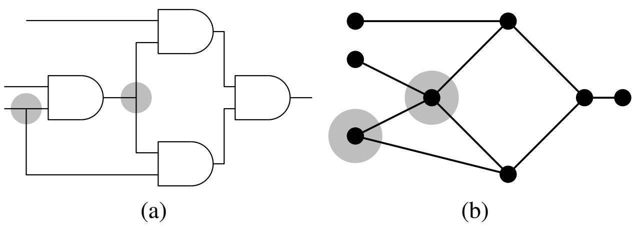

Figure 2: (a) Example circuit with fanout points highlighted (b) Equivalent AIG

suspect set, constructed to limit the total number of suspects examined across all iterations. We denote the suspect set of an iteration $i$ as $L_i$. The initial suspect set is constructed as $L_1 = R \cup F$, to include all fanout points and all latches that appear in the target state predicate. The debugging algorithm is then executed using this suspect set, returning a set of solutions $S_1$. A new suspect set $L_2$ is computed from $S_1$ and used in the subsequent iteration. In general, after iteration $i$, the new suspect set $L_{i+1}$ is computed as shown in Eq. 1 below.

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

where $fanin(l)$ denotes the set of all fanin for a location $l$. Note that $L_{i+1}$ contains the fanin of every solution found in iteration $i$ minus the union of all suspect sets for previous iterations. In other words, $L_{i+1}$ does not include any locations used as suspects in a previous iteration. This both ensures that no location is a suspect in multiple iterations, and guarantees that the algorithm terminates.

The reasoning behind this approach is fairly intuitive. If a location $l$ is a solution, it means that $l$ can be replaced by a different Boolean function to make an $\mathcal{S}$-state reachable. It may also be possible to do the same at one of its fanins, so the algorithm must check if its fanins are also solutions. On the other hand, if $l$ is not a solution then there is no way to modify the design at location $l$ and fix the error. In some cases, this may imply that elements in the set $fanin(l)$ also are not solutions. In particular, if $l$ is not a solution, $l' \in fanin(l)$, and $l' \notin (F \cup R)$, then $l'$ is not a solution. This important result is proven in the next subsection. From this result and the construction of the initial suspect set, the algorithm does not need to check if such a location is a solution. Therefore, none of these locations are included in the suspect set.

The steps of the approach are shown in Algorithm 1. In that description, algorithm UNREACHABILITY is as described in section 2.2. Lines 1 and 2 construct the sets $R$ 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 set of suspects for the next iteration is constructed on line 6 according to Eq. 1. Finally, line 9 constructs the solution set from the solution sets of each iteration, which is returned on line 10.

**Algorithm 1** UNREACHABILITYENHANCED$(C, \mathcal{S})$

```
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$(C, \mathcal{S}, L_i) \neq \emptyset$ **do**
6:      $L_{i+1} = \{l \in S_i : fanin(l)\} \setminus \bigcup_{j=1}^{i}(L_i)$
7:      $i = i + 1$
8:  **end while**
9:  $S = \bigcup_{j=1}^{i} S_j$
10: **return** $S$
```

## 3.2 Soundness and Completeness

In the previous subsection, the approach and the rationale behind the construction of the suspect sets is described. This subsection elaborates on these details and proves the soundness and completeness of Algorithm 1.

Towards the goal of demonstrating that the construction of the suspect sets as defined in Eq. 1 is reasonable, consider a location $l$ that is also a solution. As $l$ is a solution, the Boolean function at $l$ can be replaced by a different Boolean function to solve the problem. One can observe that this implies one of two possibilities. First, it may also be possible to replace one of its fanins with a different Boolean function to correct the problem, and therefore one or more of its fanin locations may also be solutions. This occurs if the needed change at $l$ is equivalent to making a change at only one of its fanin locations. Alternatively, it may not be possible to correct the problem at one of the fanins, and therefore $l$ is a solution but no element of $fanin(l)$ is. This occurs if there is no way to modify the fanin locations of $l$ individually to correct the issue. As a result, the fact that $l$ is a solution does not imply whether or not the elements of $fanin(l)$ are solutions. It is necessary to include these locations in the suspect set of a subsequent iteration to prove whether or not they are solutions.

Conversely, if a location $l'$ is not a solution then this may imply that some elements of $fanin(l')$ are not solutions. Consider a location $l \in fanin(l')$. If $l$ has other fanouts besides $l'$, it may be possible for $l$ to be a solution even if all of its fanouts are not. This case can occur if multiple fanouts of $l$ need to be simultaneously 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 fanouts are. However, if $|fanout(l)| = 1$, $l \notin R$, and the single fanout of $l$ is not a solution, then $l$ also is not a solution. The following lemma formalizes this notion.

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

*Proof.* 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.

However, 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 $R$. □

This demonstrates the rationale behind constructing the initial suspect set as $F \cup R$. The set $F$ includes every location $l$ with $|fanout(l)| > 1$. As a result, every $l \notin L$ satisfies $l \notin R$ and $|fanout(l)| \leq 1$, meaning that Lemma 1 can remove it from consideration if the single element of $fanout(l)$ is not a solution. This allows the algorithm to remove a large number of locations from consideration without passing them to UNREACHABILITY. Essentially, the initial suspect set is constructed to cover all of the cases that the lemma cannot.

We now turn our attention to proving the soundness and completeness of Algorithm 1. In this context, soundness implies that every location it returns is indeed a solution. Completeness requires that it also returns every solution. The following theorem shows that the approach is sound. We omit its proof because it follows trivially from the soundness of UNREACHABILITY as proven in (Berryhill and Veneris 2016).

**Theorem 1.** *Every location in $S$ is a solution.*

Since the set $S$ only includes locations identified as solutions by UNREACHABILITY, every location in $S$ is a solution. Because $S$ is the set of solutions in Algorithm 1, Theorem 1 proves the soundness of the algorithm. Theorem 2 below proves that the algorithm is complete, which follows from the construction of the initial suspect set and Lemma 1.

**Theorem 2.** *When Algorithm 1 terminates, $S$ includes every solution.*

*Proof.* The initial suspect set is $F \cup R$. Since UNREACHABILITY 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 $l \notin R$ 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, the algorithm constructs a new suspect set 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 UNREACHABILITY. Therefore, by the completeness of UNREACHABILITY, 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 1, Theorem 2 proves that the algorithm is complete. In contrast with the previous approach, the algorithm does not require the user to specify a set of suspect locations. Since the algorithm carefully selects the suspect sets it examines, 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

Table 1: Experimental Results

| Benchmark | | | Previous Approach | | | Algorithm 1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| benchmark | #gates | #latches | $\|L\|$ | #sol | time (s) | #sol | #iter | $\|\bigcup L_i\|$ | time (s) | $\|\bigcup L_i\|/\|L\|$ | speedup |
| mrisc_core | 8165 | 1328 | 9573 | 18 | 276.9 | 18 | 6 | 1708 | 6.1 | 17.8% | 45.6x |
| design1 | 1070 | 147 | 1208 | 9 | 11.4 | 9 | 4 | 232 | 0.4 | 19.2% | 25.9x |
| divider | 3555 | 360 | 3915 | 38 | 419.6 | 38 | 3 | 1056 | 12.2 | 27.0% | 34.3x |
| spi | 1009 | 132 | 1156 | 23 | 7.9 | 23 | 7 | 246 | 0.7 | 21.3% | 11.6x |
| wb | 390 | 61 | 451 | 193 | 10.0 | 193 | 8 | 237 | 0.5 | 52.5% | 19.7x |
| usb_core | 4856 | 534 | 5545 | 6 | 631.4 | 6 | 3 | 1140 | 3.5 | 20.6% | 194.3x |
| ac97_ctrl | 12607 | 2325 | 14967 | 13 | 496.4 | 13 | 4 | 2697 | 17.9 | 18.0% | 27.8x |
| **GEOMEAN** | | | | | | | | | | 23.4% | 33.7x |
| **MEDIAN** | | | | | | | | | | 20.6% | 27.8x |

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 1 does not provide this functionality, it is 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.

## 4 Experimental Results

All results presented in this section are run on a single core of an i5-3570K 3.4 GHz workstation with 16GB of RAM. The proposed algorithm is developed on top of a reference implementation of PDR (Bradley 2011). For comparison the approach of (Berryhill and Veneris 2016) is implemented using the same PDR implementation. Six designs from OpenCores (OpenCores.org 2007) and one commercial design from an industrial partner are utilized as benchmarks. Each problem instance is created by injecting a design error that makes at least one 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 an industry setting. When executing the approach of (Berryhill and Veneris 2016), the suspect set $L$ is chosen to include every location in the circuit.

Table 1 shows comprehensive results. The first three columns show the name of the problem instance, the number of gates in the design, and number of latches, respectively. The number of gates and latches are derived from the AIG representation of the design. The next three columns show the size of the suspect set, number of solutions found, and runtime for the previous approach. The remaining five columns relate to Algorithm 1. 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 the previous approach, respectively.

It can be seen that both approaches always find the same set of solutions, as expected. Across all experiments, the proposed approach offers a geometric mean speedup of 33.7x with a median of 27.8x. The proposed approach is able to safely ignore a majority of all design locations.

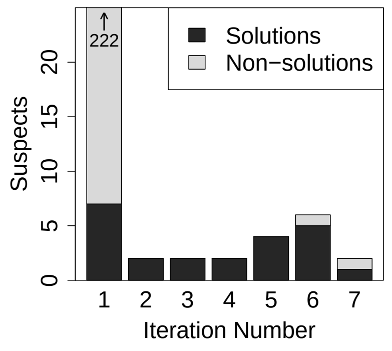

Figure 3: Solutions and non-solutions per iteration for `spi`

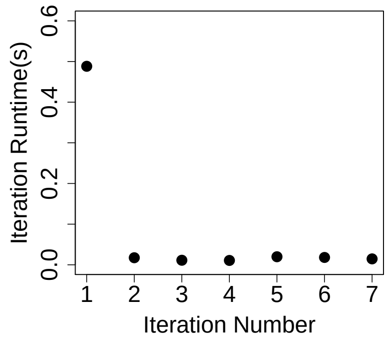

Figure 4: Runtime for each iteration for `spi`

Across all experiments, the proposed approach is able to use Lemma 1 to ignore a geometric mean of 76.6% of all locations, and a median of 79.4%. 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 3 plots the number of solutions and non-solution suspects for each iteration for the design `spi`. 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

Table 2: Number of suspects and solutions in each iteration

| benchmark | $\lvert S_1\rvert/\lvert L_1\rvert$ | $\lvert S_2\rvert/\lvert L_2\rvert$ | $\lvert S_3\rvert/\lvert L_3\rvert$ | $\lvert S_4\rvert/\lvert L_4\rvert$ | $\lvert S_5\rvert/\lvert L_5\rvert$ | $\lvert S_6\rvert/\lvert L_6\rvert$ | $\lvert S_7\rvert/\lvert L_7\rvert$ | $\lvert S_8\rvert/\lvert L_8\rvert$ |
|---|---|---|---|---|---|---|---|---|
| mrisc_core | 4/1688 | 4/4 | 2/2 | 3/4 | 3/6 | 2/4 | - | - |
| design1 | 4/225 | 3/3 | 2/2 | 0/2 | - | - | - | - |
| divider | 10/1028 | 10/10 | 18/18 | - | - | - | - | - |
| spi | 7/229 | 2/2 | 2/2 | 2/2 | 4/4 | 5/6 | 1/2 | - |
| wb | 33/76 | 33/34 | 33/33 | 34/34 | 4/4 | 8/8 | 16/16 | 32/32 |
| usb_core | 3/1136 | 1/2 | 1/1 | 1/1 | - | - | - | - |
| ac97_ctrl | 5/2689 | 2/2 | 2/2 | 2/2 | 2/2 | - | - | - |

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 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 each design. It demonstrates that in most cases, the initial suspect set contains few solutions, meaning that very few suspects are considered in subsequent iterations. The design `wb` appears to be the only exception. In this case, 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 19x 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 Figure 4, 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 unsurprising, 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 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 5 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 towards 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

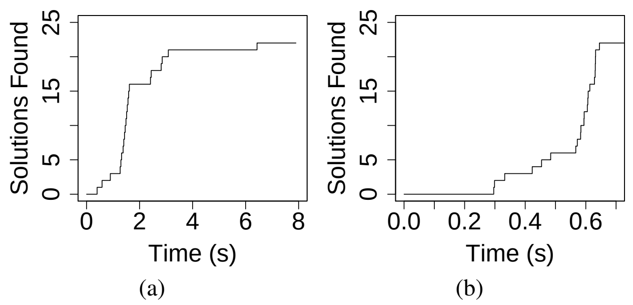

Figure 5: Solutions found for `spi` vs. running time for (a) the previous approach (b) Algorithm 1

time to prove no further solutions exist before terminating.

Conversely, Figure 5(b) shows that Algorithm 1 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.

## 5 Conclusion

This work presents an algorithm to diagnose errors that cause unreachable states. It is complete by nature and returns the complete solution set of the problem. It improves upon the previous technique by ignoring a large portion of the design locations that are provably not solutions. Empirical results confirm that a majority of design locations are safely ignored resulting in a substantial runtime improvement.

## References

Berryhill, R., and Veneris, A. 2015a. Automated rectification methodologies to functional state-space unreachability. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, 1401–1406.

Berryhill, R., and Veneris, A. 2015b. Diagnosing unreachable states using property directed reachability. In *Proceedings of the 2015 Workshop on Constraints in Formal Verification*, CFV '15.

Berryhill, R., and Veneris, A. 2016. 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.

Bradley, A. 2011. Sat-based model checking without unrolling. In *Intl Conf. on Verification, Model Checking, and Abstract Interpretation*, 70–87.

Brummayer, R., and Biere, A. 2006. 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.

Chang, K.-H.; Markov, I.; and Bertacco, V. 2007. Automating post-silicon debugging and repair. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, 91–98.

Fey, G.; Staber, S.; Bloem, R.; and Drechsler, R. 2008. Automatic fault localization for property checking. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27(6):1138–1149.

Foster, H. 2008. Assertion-based verification: Industry myths to realities (invited tutorial). In *Intl Conference on Computer-Aided Verification (CAV)*, 5–10.

Foster, H. 2011. From volume to velocity: The transforming landscape in function verification. In *Design Verification Conference*.

Huang, S.-Y., and Cheng, K.-T. 1998. *Formal Equivalence Checking and Design DeBugging*. Norwell, MA, USA: Kluwer Academic Publishers.

OpenCores.org. 2007. http://www.opencores.org.

Smith, A.; Veneris, A.; Ali, M. F.; and Viglas, A. 2005. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst* 24(10):1606–1621.