Automated Rectification Methodologies to Functional State-Space Unreachability

Ryan Berryhill[1], Andreas Veneris[1,2]

***Abstract*— In the modern design cycle, significant manual resources are dedicated to fix a design when verification shows that a state is not reachable. Today there is little automation to aid an engineer in understanding why a state is not reachable and how to correct it. This paper presents a novel methodology that automates this task. In detail, a process that involves intertwined steps of state approximation, reachability analysis and traditional debugging is developed to identify design locations where fixes can be applied so the target state becomes reachable. An initial formulation identifies such error locations that, when corrected, can make the target state reachable directly from the existing reachable set of states. This is later extended for the cases where more than one state transition is required to reach an unreachable state from the existing reachable set. Empirical results on industrial level designs show a performance which is an order of magnitude faster than the state-of-the-art confirming the practicality of the proposed automated methodology.**

I. Introduction

Functional verification has grown to be the major bottleneck in modern hardware design taking more than 70% of the overall design effort [1]. Debugging, the task of localizing the error source, accounts for a hefty 60% of the verification cycle [2]. Most verification and debugging tasks have been automated or semi-automated. Despite these advances, once static or dynamic verification shows that a state is not reachable, identifying the root cause of the failure remains a predominantly manual effort with little automation available.

Traditionally, when verification detects an error such as an observation signal value mismatch, a scoreboard discrepancy, or a firing assertion, an error trace (counter-example) is returned that demonstrates that failure. This error trace is later used by a debugging tool [3]–[6] to identify the root cause of the problem so the engineer can fix it. On the other hand, when verification fails because a state is not reachable, an error is clearly detected but no such error trace exists, in the traditional definition of the term, to guide automated debugging. As a result, correcting the root cause for an unreachable state today remains a largely manual process that consumes significant resources and engineering effort. In the context of coverage analysis, the work in [7] discusses the problem of unreachable code. While unreachable code may be a symptom of an unreachable state(s) and vice versa, in practice they often manifest themselves separately. Therefore, automated techniques to aid the engineer in fixing a design when a state is shown to be unreachable are of paramount importance to reduce the verification burden and improve the design cycle.

Towards this direction, we present an automated methodology to identify suspect locations where fixes can be implemented so that a design reaches a target unreachable state. Initially, the methodology tackles unreachable states that can be reached within one cycle (*i.e.,* transition) from the existing reachable set of states. This is done by utilizing formal techniques to compute an over-approximation of the reachable states for a specific operational design cycle. This over-approximation essentially provides a set of constraints for that particular cycle that is debugged by modifying the input set of a traditional Boolean satisfiability (SAT) debugger. Due to the inherent nature of the state-space approximation, spurious solutions may be present. These solutions are identified and discarded in an effort to refine the input constraint model and increase the accuracy of the approximation. In practice, the accuracy of the approximation increases rapidly and spurious solutions occur infrequently. These intertwined steps of state-space approximation and debugging are repeated increasing the number of cycles from the initial set of states until a solution is found, that is, a design location where a fix can be performed so the target state becomes reachable.

This automated methodology is later extended to tackle cases where multiple transitions are required from the reachable set of states to reach the target unreachable state. Furthermore, as the solution space may grow significantly larger with more iterations, the algorithms are enhanced with a set of techniques that provide a configurable tradeoff between run-time and resolution.

Experiments on sequential designs with injected design errors confirm the performance and validity of the approach. Furthermore, results demonstrate that the approximations used in the algorithm improve in accuracy very rapidly, making spurious solutions infrequent and inexpensive to detect. The initial technique provides an average speedup of 2x when compared to the current state-of-the-art debugging technique. An additional optimization is presented that achieves an impressive 26x speedup.

The remainder of this paper is organized as follows. Background on reachability analysis and automated debugging is presented in Section II. Section III describes the initial approach. Section IV presents an extension to the initial approach. Section V describes a performance optimization. Section VI presents experimental results, and finally, Section VII concludes the paper.

II. Preliminaries

A. Notation

The following notation is used throughout this paper. Given a sequential circuit C, the set of primary input, primary output, and state elements (flip flops) of C are denoted by $X = \{x_1, x_2, ..., x_{|X|}\}$, $Y = \{y_1, y_2, ..., y_{|Y|}\}$, and $S = \{s_1, s_2, ..., s_{|S|}\}$, respectively. In an iterative logic array (ILA) [8] representation of the circuit, superscripts distinguish between clock cycles where $S^i = \{s_1^i, s_2^i, ..., s_{|S|}^i\}$ represents the values or the actual state elements in cycle i. Similar notation is used for the primary input and output. We let the set of initial states for C be $I(S)$.

The transition relation of C is denoted as $T(S^i, S^{i+1}, X^i, Y^i)$. It evaluates to `1` if and only if given current state S^i, applying X^i to the primary input of C it yields next state S^{i+1} and primary output Y^i. We say that a state is *k-reachable* if there is a sequence of Boolean values that can be applied to the primary input to cause the circuit to reach the state in k or fewer cycles. If there is a value of k for which a state is k-reachable, then we also say that the state is *reachable*. This paper denotes the set of k-reachable states as R_k. Finally, we let $\mathcal{S}$ be the target unreachable state.

B. Background

In the context of this work, reachability analysis refers to the process of approximating the set of k-reachable states for a given value of k. Calculating the set R_k is intractable, but efficient algorithms exist to over-approximate it [9], [10]. In particular, this work does not introduce new techniques related to reachability analysis. It deals exclusively with the problem of how to fix an erroneous design when a verification tool shows that a particular state is unreachable in

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

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

978-3-9815370-4-8/DATE15/©2015 EDAA

violation of its specification. To tackle this problem, it utilizes three by-product aspects of the work in [9], namely reachability analysis, reachability checking, and approximation strengthening. Due to their relevance in this paper, we describe them in greater detail.

The work in [9] performs reachability analysis directed towards proving a given safety property. This is accomplished by constructing a sequence of sets $F = \langle F_0, F_1, F_2, ..., F_k \rangle$ where $F_0 = I(S)$ and each F_i is an over-approximation of the set of i-reachable states (*i.e.* $R_i \subseteq F_i$). Each set F_i can be represented by a propositional formula over the state elements of the circuit [9]. In this paper, given some state S^i, we define function $F_i(S^i) = 1$ iff state S^i is in the set F_i. Given a state and a cycle i, reachability checking determines if the state is in the set R_i. Finally, strengthening improves the accuracy of F_i by returning a smaller over-approximation that excludes some states that can be proven not to be i-reachable. Given a particular state that is not i-reachable, strengthening returns clauses that can be conjoined to F_i to reduce its size [9]. The improved approximation excludes the given state, and may also exclude additional states proven not to be i-reachable.

The work presented here also iteratively utilizes the SAT-based automated debugging framework from [3]. In detail, [3] identifies suspect locations that when corrected, fix the erroneous behavior exposed by a counter-example. Let X^i denote the primary input values from the counter-example in cycle i and allow Y^i to denote the reference primary output logic values in cycle i, as defined earlier. Let $B = \{b_1, ..., b_{|B|}\}$ denote the RTL blocks in the circuit, where the output of block b_j in cycle i is b_j^i. An enhanced transition relation, $T_{en}(S^i, S^{i+1}, X^i, Y^i, e)$ is constructed with added error-select lines $e = \{e_1, ..., e_{|B|}\}$. If $e_i = 0$, the behavior of b_i is unchanged, while setting $e_i = 1$ replaces b_i^j with a free variable w_i^j for all values of j.

Additional input and output constraints are then derived from the counter-example to set the primary input to the values from the counter-example and to force the primary output to the reference logic values for cycle i, respectively. Further, constraint $I(S^0)$ ensures that the circuit begins at a particular initial state. Finally, the number of simultaneously-active error-select lines is limited to a user-specified value N with a cardinality constraint Φ_N.

As such, for a k-cycle counter-example, the problem encoding is:

$$D = I(S^0) \wedge \bigwedge_{i=0}^{k} \Big(T_{en}(S^i, S^{i+1}, X^i, Y^i, e) \Big) \wedge \Phi_N \qquad (1)$$

Where X^i represents the primary input values from cycle i of the counter-example and Y^i is set to the reference primary output values for cycle i. Each satisfying assignment of Eq. 1 corresponds to an N-tuple of suspect locations that can be corrected to fix the erroneous behavior of the counter-example.

We conclude this section with a *brute-force* automated approach to debug a design for an unreachable state. As to the best of our knowledge there is no prior work for us to compare, we will later contrast this approach with the methodology developed in this paper. Such a brute-force approach can use the state-of-the-art methodology from [3] to generate a debugging problem aimed at correcting a design error that causes an unreachable state. It creates an ILA of i cycles which is later constrained using the initial set of states and by placing the target unreachable state $\mathcal{S}$ as a final state constraint. In essence, this approach annotates Eq. 1 as follows:

$$\begin{aligned} B(i, \mathcal{S}) = I(S^0) \wedge \bigwedge_{j=0}^{i} T_{en}(S^j, S^{j+1}, X^j, Y^j, e) \\ \wedge (S^{i+1} = \mathcal{S}) \wedge \Phi_N \end{aligned} \qquad (2)$$

By solving the constraint satisfaction problem in Eq. 2 for increasing values of $i \leq k$, the method essentially searches for error locations where fixes can be performed to make the target state $(i+1)$-reachable. By construction, the method is exhaustive, that is, it will return all solutions to the problem. On the other hand, as the state-space explodes with increasing values of i, its performance may deteriorate as the SAT solver gradually explores a much larger solution space. This fact motivates for the development of novel methodologies tailored to the problem of state unreachability.

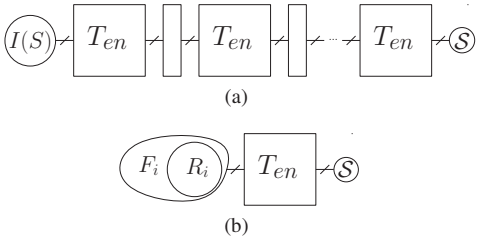

Fig. 1. Model of (a) $B(i, \mathcal{S})$ (b) $U(i, \mathcal{S})$

III. Uni-Cycle Unreachability

This section presents an algorithm to localize bugs that cause a design to have unreachable states. Given an erroneous circuit C and an unreachable target state $\mathcal{S}$, the algorithm finds suspect locations that can be changed to make $\mathcal{S}$ reachable with one transition from some already-reachable state.

The algorithm consists of a sequence of iterations, each of which models and debugs a single state transition from an already-reachable state to $\mathcal{S}$. As calculating the exact reachable set of states is an intractable problem, at each step of the algorithm an over-approximation is utilized to model the potential set of reachable states. Spurious solutions that arise from the use of an approximation are detected and discarded. For simplicity, the initial formulation presented in this section only identifies error locations for which the target can be made reachable directly from an already-reachable state. The next section extends the method to handle the cases where other unreachable states must be reached prior to reaching the target state.

Specifically, the i-th iteration searches for solutions that may make the target state $(i+1)$-reachable. Each iteration consists of two steps: *reachability analysis* and *debugging*. Reachability analysis calculates an initial approximation F_i of the set of i-reachable states using the reachability analysis procedure of [9], described in Section II-B, where $\mathcal{S}$ is used as the safety property to prove. For the purposes of this formulation, reachability analysis can be treated as a "black box," and so the remainder of this section focuses on debugging.

The debugging step constructs a SAT-based debugging instance, with the goal of finding suspect locations that can be changed to allow for a state transition from a state in R_i to the target state $\mathcal{S}$. Towards this end, the instance utilizes a single copy of the transition relation constrained by set F_i at its input and the target state $\mathcal{S}$ at its output. Intuitively, the current set of states for the debugging instance is constrained using F_i while the next state is constrained to $\mathcal{S}$. The primary input and output variables are left unconstrained, allowing the SAT solver to find solutions for any input assignment. As such, the resulting debugging instance can be expressed as follows:

$$U(i, \mathcal{S}) = F_i(S^i) \wedge T_{en}(S^i, S^{i+1}, X^i, Y^i, e) \wedge (S^{i+1} = \mathcal{S}) \wedge \Phi_N \qquad (3)$$

Note that solutions to Eq. 3 merely indicate locations where a change can be applied to make the target state reachable. The engineer must make an informed decision as to how to implement these changes while maintaining the required functionality. As is the case for traditional debugging based on a set of counter-examples [3], a full verification step is required to confirm the correctness of the modified design.

It is instructive to compare the proposed formulation to the brute-force approach. Towards this end, Fig. 1 illustrates the models represented by Eq. 2 and Eq. 3. In solving the ILA behind $B(i, \mathcal{S})$, the SAT solver is given the freedom to set values for all the primary

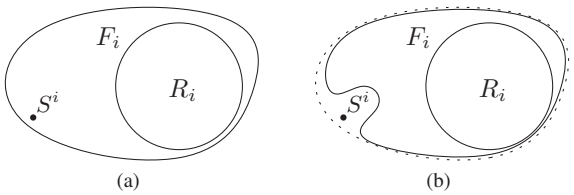

Fig. 2. Set F_i (a) initially (b) after strengthening with state $S^{i'}$

input variables and the error-select lines. By construction, the state elements (shown as slices between copies of the transition relation in Fig. 1(a)) for the ILA with $i+1$ copies of the transition relation will assume values to represent a sequence of states so that the overall problem reaches the target state $\mathcal{S}$. As noted earlier, with increasing values of i the state-space grows exponentially for the SAT solver to find proper values for the state elements to reach $\mathcal{S}$. Nevertheless, the solutions returned are exact as the method in [3] is exhaustive.

On the other hand, Fig. 1(b) depicts the formulation of $U(i, \mathcal{S})$. We observe that the proposed method is given the freedom to select candidate current states from the set F_i in its effort to reach the target state. In this context, a solution of cardinality N to this debugging instance consists of a set of N active error-select lines and a state in the set F_i. As set F_i is an approximation of the set of R_i, inherent to the method is the fact that the set of solutions returned may not be exact. This means that some solutions that satisfy Eq. 3 may not be actual solutions to the unreachability problem as expressed by Eq. 2, which is exhaustive. We therefore define a *spurious* solution as a solution that satisfies $U(i, \mathcal{S})$ but does not satisfy $B(j, \mathcal{S})$ for any value of $j \leq i$. Note, the term spurious here is distinct from the term used in a traditional abstraction/refinement context [11].

Evidently, spurious solutions may be returned when solving Eq. 3 if the chosen current state is a member of set $F_i - R_i$ and it is not i-reachable. Spurious solutions do not represent design locations that can be changed to make the target state reachable. Therefore, the algorithm must detect such cases and reject them. On the other hand, any solution for $U(i, \mathcal{S})$ when the current state is indeed i-reachable is non-spurious. Therefore, when a solution is found the reachability checking procedure of [9] is used to determine if the current state is i-reachable or not. If it is, the solution is proven to be non-spurious and it is added to the final set. Alternatively, if the current state is shown not to be i-reachable, it is discarded. When this happens, the strengthening procedure of [9] is used to strengthen the approximation F_i. At a minimum, strengthening will remove the current state that was found not be i-unreachable from F_i. This prevents the SAT solver from finding further spurious solutions with the same current state in the current iteration.

In practice, the work of [9] may remove many other states that can be proven not to be reachable resulting in a rapid increase of the accuracy of the approximation and hence our method. This is shown in Fig. 2 where state S^i has been proven to be spurious in the current iteration of the algorithm. As a result of the strengthening process, a more accurate approximation is derived shown in Fig. 2(b). An accurate approximation provides numerous benefits to the algorithm. During the current iteration it reduces the number of potential spurious solutions, and in future iterations it can both increase the accuracy of the initial approximation of the reachable set and improve the overall run-time performance.

Although the process above removes spurious solutions and strengthens the approximation, it should be noted that not all elements of $F_i - R_i$ may generate spurious solutions for $U(i, \mathcal{S})$. If a solution is derived from a current state that is not i-reachable, it may be the case that a single fix in this location can make *both* the current state and the target state $\mathcal{S}$ reachable. Although we should expect this to be a rare case, the extension in the next section handles these situations.

Pseudo code for the entire procedure is shown in Algorithm 1.

Algorithm 1 UNIUNREACHABILITY$(C, \mathcal{S}, k)$

```
1:  solutions = ∅
2:  for i in 0, 1, ..., k do
3:     F_i = REACHABILITYANALYSIS(i)
4:     U = DEBUGGINGINSTANCE(C, S, F_i)
5:     while (Solution = SAT(U)) ≠ UNSAT do
6:        if REACHABILITYCHECK(Solution, F_i) then
7:           solutions = solutions ∪ {Solution}
8:        else
9:           NewClauses = STRENGTHEN(F_i, Solution)
10:          F_i = F_i ∧ NewClauses
11:          U = U ∧ NewClauses
12:       end if
13:    end while
14: end for
15: return solutions
```

It assumes the existence of procedures REACHABILITYANALYSIS, REACHABILITYCHECK, and STRENGTHEN from the work in [9]. Further, procedure DEBUGGINGINSTANCE generates the propositional formula in Eq. 3. Line 3 of the algorithm constructs the initial approximation F_i, which is utilized to construct the debugging instance on line 4. After a solution is found by the SAT solver, line 6 attempts to prove it is non-spurious. Finally, lines 9-11 apply the strengthening procedure if the current state of the solution is found not to be i-reachable.

The theorem that follows proves the correctness of the algorithm.

Theorem 1 *In iteration i, the algorithm finds exactly the set of all solutions that make the target state $\mathcal{S}$ reachable in one cycle from an i-reachable state.*

Proof: The debugging instance of Eq. 3 uses F_i as its current state set. Throughout the iteration, F_i is updated, but $R_i \subseteq F_i$ always holds. Therefore, the current state set of the debugging instance always includes all states in R_i, implying that the algorithm finds every solution that reaches the target state and has a current state in R_i. Furthermore, solutions with a current state that is not in R_i are rejected, implying that all solutions found have a current state in R_i. This implies that it finds exactly the set of all solutions that allow for a state transition from a state in R_i to the target state $\mathcal{S}$. ■

Theorem 1 proves that the algorithm works when $\mathcal{S}$ can be reached with one transition from a state that is i-reachable. However, the target state may not be the only erroneously unreachable state. It may be the case that it can be reached through a sequence of states that are all erroneously unreachable. The following section extends the algorithm to find solutions in these cases.

IV. MULTI-CYCLE UNREACHABILITY

To obtain solutions that reach the target state $\mathcal{S}$ in more than one cycle from an already-reachable state, the approach presented here models a sequence of n state transitions that originates from a reachable state and ultimately transitions to $\mathcal{S}$. To do this, the corresponding debugging instance is expressed as follows:

$$M(i, n, \mathcal{S}) = F_i(S^i) \wedge \bigwedge_{j=i}^{i+n-1} T_{en}(S^j, S^{j+1}, X^j, Y^j, e) \wedge (S^{i+n} = \mathcal{S}) \wedge \Phi_N \quad (4)$$

The parameter i represents the number of clock cycles modeled by the approximation F_i, while n represents the *window size*, that is the number of state transitions the algorithm is allowed to "look forward" for unreachable states so that it reaches $\mathcal{S}$. Rather than finding solutions that make the target state $(i+1)$-reachable, $M(i, n, \mathcal{S})$ can return solutions that make the target state $(i+n)$-reachable. As was

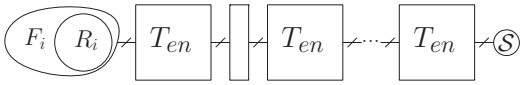

Fig. 3. Model of $M(i, n, \mathcal{S})$

the case for Eq. 3, a solution to $M(i, n, \mathcal{S})$ of cardinality N consists of N active error-select lines and a current state in the set F_i.

One may observe that the brute-force approach and the single-cycle approach are merely orthogonal special cases of Eq. 4. A window size of one makes this approach equivalent to the uni-cycle approach, that is, $M(i, 1, \mathcal{S}) = U(i, \mathcal{S})$. Conversely, using a window of $k + 1$ cycles and constraining the current state to the initial states of the circuit makes this approach equivalent to the brute-force approach as $M(0, i + 1, \mathcal{S}) = B(i, \mathcal{S})$. Again, Fig. 3 illustrates the model of $M(i, n, \mathcal{S})$ to demonstrate features shared with Eq. 2 and Eq. 3.

Due to the use of approximation, some solutions to to $M(i, n, \mathcal{S})$ may not be solutions to the unreachability problem. That is, solutions to $M(i, n, \mathcal{S})$ exist that are not solutions to $B(j, \mathcal{S})$ for any $j < i+n$. Therefore, a means of rejecting these spurious solutions is needed. Intuitively, a solution for which the current state is i-reachable is non-spurious by the same argument used in the previous section. Therefore, the extended algorithm uses the same mechanism used in Algorithm 1 to reject spurious solutions, that is when a solution is found, the current state is checked for i-reachability. If it is found not to be i-reachable, it is discarded and used to improve the approximation F_i. Otherwise the solution is added to the final set.

Pseudo code for the procedure is shown in Algorithm 2. In addition to the procedures used in Algorithm 1, it assumes the existence of a procedure MULTICYCLEDEBUGGINGINSTANCE, which generates the propositional formula of Eq. 4. In the algorithm, line 5 calculates the initial approximation of the reachable set. In iterations prior to iteration n this is simply F_0 (*i.e.* the initial states). In later iterations, it is an approximation of the $(i - n + 1)$-reachable set. Lines 7-15 essentially perform an iteration of Algorithm 1 using the n-cycle debugging instance of Eq. 4 in place of the single-cycle debugging instance of Eq. 3. Note that in iterations $0 \leq i < n$ the algorithm models only $(i+1) \leq n$ clock cycles, allowing it to still find solutions that make the target state $(i+1)$-reachable in these iterations. Further, the current state set is restricted to the set of initial states, essentially making it equivalent to the brute-force approach.

The following theorem shows the algorithm correctly returns all solutions. We omit its proof which is similar to that of Theorem 1.

Theorem 2 *The solution set of $M(i, n, \mathcal{S})$ is exactly the set of all solutions that make the target state $\mathcal{S}$ reachable n cycles after an i-reachable state.*

Theorem 2 proves the completeness of the solution set, which we now examine in more detail. By Theorem 2, after iteration k Algorithm 2 finds all solutions that make the target state reachable n or fewer cycles following a $(k-n+1)$-reachable state. This includes all solutions that make the target state reachable one cycle after a k-reachable state, and therefore the solution set of Algorithm 2 is a superset of the solution set of Algorithm 1.

In particular, the solution set of Algorithm 2 can include solutions that would be discarded in Algorithm 1. As mentioned in Section III, a solution to $U(i, \mathcal{S})$ with a current state that is not i-reachable may not be spurious if a fix at the same location can make both its current and next states reachable. By modeling multiple state transitions, it is possible for Algorithm 2 to solve such rare cases. Specifically, any solution discarded in Algorithm 1 that can make its current state reachable in fewer than n cycles from a $(k - n + 1)$-reachable state will be in the solution set of Algorithm 2.

V. PERFORMANCE OPTIMIZATION

In this section we discuss a performance-driven enhancement for the methodologies presented earlier. The enhancement is presented as a modification to Algorithm 1, but can also be applied to Algorithm 2.

Algorithm 2 MULTICYCLEUNREACHABILITY$(C, \mathcal{S}, k)$

```
1:  solutions = ∅
2:  for i in 0, 1, ..., k do
3:      n' = min(n, i + 1)
4:      t = i − n' + 1
5:      F_t = REACHABILITYANALYSIS(t)
6:      M = MULTICYCLEDEBUGGINGINSTANCE(C, S, F_t, n')
7:      while (Solution = SAT(M)) ≠ UNSAT do
8:          if REACHABILITYCHECK(Solution, F_t) then
9:              solutions = solutions ∪ {Solution}
10:         else
11:             NewClauses = STRENGTHEN(F_t, Solution)
12:             F_t = F_t ∧ NewClauses
13:             M = M ∧ NewClauses
14:         end if
15:     end while
16: end for
17: return solutions
```

Given iteration limit k, Algorithm 1 must solve $k + 1$ debugging instances. Each iteration finds a new over-approximation enlarging the set of current states to be considered. This implies that each iteration has the potential to find a larger set of solutions than previous iterations. The proposed modification simply skips the first k iterations and starts by executing the final iteration directly. It computes $F_0, F_1, ..., F_k$ using the reachability analysis procedure from [9] without strengthening any of the intermediate approximations. It then proceeds to solve $U(k, \mathcal{S})$, rejecting potentially spurious solutions and strengthening the approximations as done in Algorithm 1.

While this approach returns the same solution set as the original algorithm, it may sacrifice resolution. In particular, a solution that Algorithm 1 finds in iteration i but not in any earlier iterations can make the target state reachable in a minimum of $i + 1$ cycles. The modified approach will find the same solution, but it will not indicate the minimum number of cycles in which the solution can reach the target state. The benefit of course is that the algorithm solves only one problem instance when compared to the $k + 1$ incremental instances of the original algorithm. Hence, this modification presents a tradeoff between run-time and resolution.

Further, as the optimized approach does not strengthen the intermediate approximations, reachability analysis may compute inaccurate approximations. This can produce more spurious solutions and increase the run-time of the reachability checking procedure of [9], making it more expensive to identify potentially spurious solutions. Nevertheless, empirical results presented in the next section demonstrate that strengthening tends to rapidly improve the accuracy of the approximations removing such spurious solutions quickly. As a result, the modified algorithm exhibits a run-time performance which is an order of magnitude better than that of the original approach.

VI. EXPERIMENTAL RESULTS

All empirical results presented in this section are run on a single core of an i5-3570K 3.4 GHz workstation with 16GB of RAM using a timeout of 14400 seconds. The presented algorithms are implemented using a state-of-the-art SAT-based debugger [3] with a Verilog frontend and MINISAT v2.2.0 [12] as the underlying SAT solver. Reachability analysis/checking and strengthening are based on the implementation of property directed reachability [13] within ABC release 1.01 [14]. Five designs from OpenCores [15] and one commercial design from an industrial partner are utilized as benchmarks. Each problem instance is created by injecting a design error such as complementing conditions in if-statements, introducing incorrect state transitions, changing operators in expressions, etc. These are typical design errors observed in industry. Each design error is chosen such that it makes at least one state unreachable.

TABLE I
EXPERIMENTAL RESULTS

Benchmark		Brute-Force		Uni-Cycle			Optimized Uni-Cycle			Multi-Cycle $n=5$			Optimized Multi-Cycle $n=5$		
benchmark	k	time (s)	#solutions	time (s)	#solutions	speedup	time (s)	#solutions	speedup	time (s)	#solutions	speedup	time (s)	#solutions	speedup
wb	10	206	246	29	2	7.1x	3.7	2	55.3x	211	246	1.0x	19	246	10.5x
wb	25	4020	246	70	2	57.1x	3.7	2	1078.0x	510	246	7.9x	20	246	203.1x
wb	50	-	-	139	2	-	3.8	2	-	1012	246	-	20	246	-
wb	100	-	-	277	2	-	3.8	2	-	2024	246	-	20	246	-
wb	200	-	-	554	2	-	4.1	2	-	4056	246	-	20	246	-
mrisc_core	10	100	8	408	8	0.2x	190	8	0.5x	743	8	0.1x	130	8	0.8x
mrisc_core	25	305	8	751	8	0.4x	665	8	0.5x	2223	8	0.1x	829	8	0.4x
mrisc_core	50	1069	8	1956	8	0.5x	1095	8	1.0x	5606	8	0.2x	951	8	1.1x
mrisc_core	100	13699	8	10250	8	1.3x	1394	8	9.8x	-	8	-	1924	8	7.1x
mrisc_core	200	-	8	-	8	-	2117	8	-	-	8	-	3699	8	-
design1	10	69	30	39	21	1.8x	4.2	21	16.5x	75	25	0.9x	6.7	25	10.3x
design1	25	268	30	95	21	2.8x	4.3	21	61.9x	186	25	1.4x	7.4	25	36.1x
design1	50	809	30	189	21	4.3x	4.7	21	173.9x	389	25	2.1x	7.8	25	103.6x
design1	100	3003	30	394	21	7.6x	4.8	21	623.5x	847	25	3.5x	8.0	25	375.5x
design1	200	-	30	877	21	-	5.5	21	-	1840	25	-	8.4	25	-
usb_core	10	160	28	742	27	0.2x	122	27	1.3x	796	27	0.2x	262	27	0.6x
usb_core	25	769	28	-	-	-	234	-	3.2x	-	-	-	-	-	-
divider	10	39	52	35	4	1.1x	3.7	4	10.8x	45	25	0.9x	4.6	25	8.6x
divider	25	117	52	77	4	1.5x	3.7	4	31.7x	105	25	1.1x	4.7	25	24.6x
divider	50	258	52	146	4	1.8x	3.8	4	67.1x	205	25	1.3x	4.6	25	55.4x
divider	100	583	52	283	4	2.1x	3.8	4	152.9x	404	25	1.4x	4.7	25	123.8x
divider	200	1413	52	561	4	2.5x	3.9	4	365.7x	806	25	1.8x	4.8	25	291.9x
spi	10	14	22	14	22	1.0x	1.8	22	7.8x	24	22	0.6x	2.0	22	7.0x
spi	25	64	22	35	22	1.8x	2.1	22	30.6x	50	22	1.3x	2.1	22	29.5x
spi	50	267	22	70	22	3.8x	2.7	22	99.5x	92	22	2.9x	2.3	22	116.6x
spi	100	1399	22	147	22	9.5x	3.4	22	410.4x	186	22	7.5x	2.5	22	546.2x
spi	200	-	22	334	22	-	3.0	22	-	411	22	-	3.5	22	-
AVERAGE						**2.0x**			**26.6x**			**1.1x**			**20.8x**

Table I shows comprehensive results. All experiments are constrained with error cardinality $N = 1$. The first column contains the name of the problem instance, while the second shows the number of cycles for which solutions are pursued. The next two columns show the run-time and number of solutions found using the brute-force approach. The next three columns show the run-time, number of solutions found, and speedup (relative to brute-force) using the uni-cycle approach. The following nine columns show similar data for the optimized uni-cycle approach, the unoptimized multi-cycle approach and the optimized multi-cycle approach, respectively.

Note that the optimized and unoptimized approaches always find the same solutions. However, over the entire set of experiments, the optimized uni-cycle approach provides a speedup of 21.2x when compared to the uni-cycle approach. At $k = 10$, the speedup is only 6.1x, demonstrating that the optimization is more effective for larger values of k. Similarly, optimizing the multi-cycle approach gives a 29.6x speedup, reduced to 7.9x at $k = 10$.

Fig. 4 plots the number of solutions found by each approach using $k = 10$. The uni-cycle approach returns on the average 37% of the complete solution set of the brute-force approach, while the multi-cycle approach with $n = 5$ finds an average of 91%. The algorithm finds a particularly useful subset of the complete solution set, that is the subset of all solutions that require reaching n or fewer non-k-reachable states in order to reach a target state. It is expected that some of the unreachable states may be intended to remain invalid by the specification. That is, the design being debugged is expected to satisfy some invariant properties required by the specification. Therefore, limiting the solution set using the window size n is a desirable feature, as it may exclude solutions that are likely to violate these invariants. Modifying our formulation to automatically exclude solutions that violate specification invariants is a target of future work.

For all designs except for `wb` and `divider`, the uni-cycle approach proved sufficient to find the solution corresponding to the actual location where the error was introduced. This confirms the intuition that limiting the window size n is a desirable feature. Furthermore, using the multi-cycle approach with $n = 5$ proved sufficient to find this solution for every experiment.

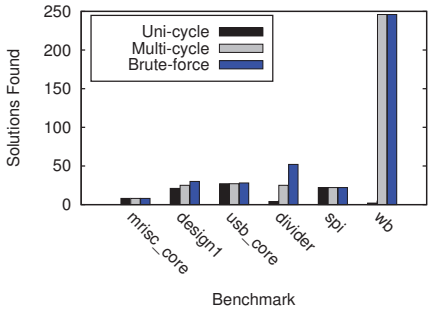

Fig. 4. Number of solutions found for each approach ($k = 10$)

Notice in particular that the algorithm finds very few solutions for `divider` and `wb` with the uni-cycle approach. This implies that the design errors in these circuits make multiple states unreachable, and most of the debugging solutions can only reach the target state through a sequence of unreachable states. Fig. 5 plots the number of solutions found for `wb` and `divider` using different window sizes. It can be seen that `wb` has 244 more solutions at $n = 5$ than at $n = 4$. This suggests that correcting the design error will result in reaching a sequence of four unreachable states prior to reaching the target state. Conversely, `divider` exhibits steady and constant growth in the number of solutions with increasing n and then it plateaus. This is because the design error occurs in a pipelined portion of the design. Increasing the window size essentially allows the algorithm to find error locations in earlier pipeline stages.

Fig. 6 shows the run-time of the brute-force and uni-cycle approaches against k for the design `spi`. It can be seen that with increasing k, the brute-force approach appears to exhibit exponential run-time growth. Conversely, the presented approach appears to

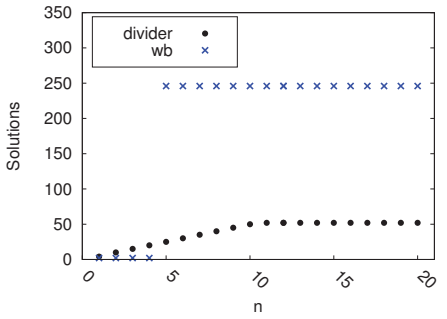

Fig. 5. Solutions vs. n for `divider` and `wb` ($k = 50$)

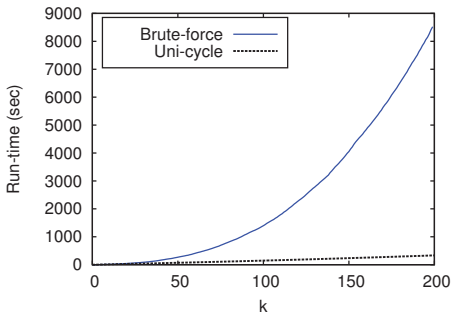

Fig. 6. Run-time vs. k for `spi`

exhibit linear run-time growth for this design. This confirms the effectiveness of our method.

Table II shows the number of spurious solutions rejected for the designs `mrisc_core` and `spi`. The first two columns show the design and the value of k used. The next four columns show the number of spurious solutions rejected for the unoptimized uni-cycle approach, unoptimized multi-cycle approach ($n = 5$) optimized uni-cycle approach, and optimized multi-cycle approach ($n = 5$), respectively. As explained earlier, in theory, the algorithm may reject a non-spurious solution that also makes its unreachable current state reachable. Across all experiments this rare case never occurred.

It can be seen that earlier iterations find more spurious solutions than later ones. This suggests that the approximated reachable sets become sufficiently accurate to prevent most spurious solutions in a relatively small number of iterations. Obviously, this is dependent on the difficulty of approximating the reachable state space for individual designs, as evidenced by the wide variation in the number of spurious solutions found between designs.

Notice that after 25 iterations, the algorithm finds very few spurious solutions for `spi`. This suggests that the reachable set is approximated with sufficient accuracy to prevent many spurious solutions after this point. In particular, it is likely that few states require more than 25 cycles to be reached in this design. Furthermore, note the difference in spurious solutions found between `spi` and `mrisc_core`. The algorithm continues to find many spurious solutions up to iteration 200 for `mrisc_core`, suggesting that its reachable set is more difficult to approximate. This further explains the difference in run-time behavior observed between the two designs.

A similar pattern appears with the optimized approaches, where the run-time for `mrisc_core` grows substantially with increasing values of k, but for `spi` remains near-constant. This is also explained by the nature of the reachable state sets for these designs. Since the set for `spi` is relatively well-approximated after 25 iterations, the optimized algorithm solves a very similar problem at $k = 25$ and at $k = 100$. However, for `mrisc_core`, the problem is substantially more difficult at $k = 100$ than at $k = 25$.

TABLE II
SPURIOUS SOLUTIONS

		Unoptimized		Optimized	
benchmark	k	uni-cycle	multi-cycle	uni-cycle	multi-cycle
mrisc_core	10	1706	2645	121	385
mrisc_core	25	1869	2953	206	199
mrisc_core	50	2219	3221	125	212
mrisc_core	100	3014	-	170	101
mrisc_core	200	-	-	128	100
spi	10	235	464	29	63
spi	25	379	599	32	11
spi	50	437	624	28	11
spi	100	487	674	33	6
spi	200	587	774	11	6

This is also the reason the optimized approach is able to achieve such large speedups in certain cases. After the reachable set of states stops expanding significantly, increasing k only negligibly impacts the run-time of the optimized algorithm. However, the brute-force approach always solves a much larger problem when k increases, and the speedup can therefore become substantial in these cases.

VII. CONCLUSION

This work presents an algorithm to localize bugs that manifest themselves as unreachable states. This is done by combining an approximated reachability analysis procedure with state-of-the-art SAT-based debugging. An optimization is also presented to improve performance. Experimental results confirm the effectiveness and practicality of the presented approach against the state-of-the-art.

REFERENCES

[1] H. Foster, "From volume to velocity: The transforming landscape in function verification," in *Design Verification Conference*, 2011.
[2] ——, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Intl Conference on Computer-Aided Verification (CAV)*, 2008, pp. 5–10.
[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.
[4] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design DeBugging*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.
[5] K.-H. Chang, I. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, Nov 2007, pp. 91–98.
[6] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 6, pp. 1138–1149, June 2008.
[7] H.-Z. Chou, K.-H. Chang, and S.-Y. Kuo, "Facilitating unreachable code diagnosis and debugging," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 485–490.
[8] H. Mangassarian, A.Veneris, S.Safarpour, M.Benedetti, and D.Smith, "A performance-driven qbf-based on iterative logic array representation with applications to verification, debug and test," in *Intl Conf. on CAD*, 2007.
[9] A. Bradley, "Sat-based model checking without unrolling," in *Intl Conf. on Verification, Model Checking, and Abstract Interpretation*, 2011, pp. 70–87.
[10] M. L. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation," in *Proceedings of the International Workshop on Logic and Synthesis*, ser. IWLS '06, 2006, pp. 172–179.
[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.
[12] N. Eén and N. Sörensson, "An extensible SAT-solver," 2003, pp. 502–518.
[13] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, TX: FMCAD Inc, 2011, pp. 125–134.
[14] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification, Release 1.01." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
[15] OpenCores.org, "http://www.opencores.org," 2007.