Managing Verification Error Traces with Bounded Model Debugging

Sean Safarpour[1] Andreas Veneris[2] Farid Najm[2]

Abstract— **Managing long verification error traces is one of the key challenges of automated debugging engines. Today, debuggers rely on the iterative logic array to model sequential behavior which drastically limits their application. This work presents Bounded Model Debugging, an iterative, systematic and practical methodology to allow debuggers to tackle larger problems than previously possible. Based on the empirical observation that errors are excited in temporal proximity of the observed failures, we present a framework that improves performance by up to two orders of magnitude and solve 2.7× more problems than a conventional debugger.**

I. Introduction

Over the past decade, the cost and time of VLSI verification and debugging has increased exponentially. Today, verification takes up to 70% of the design time with as much as half of this effort attributed to manual debugging. As a result, automated and scalable debugging methodologies are needed to aid engineers to efficiently localize the error sources.

In general, debugging techniques are used to localize the error source once verification identifies its presence. The inputs of a debugger are the erroneous circuit, an error trace composed of an input vector sequence with initial state values to reproduce the failure, and a corresponding correct output vector sequence [1]. Operating at the gate-level, module-level or Register Transfer Level (RTL), all components (gates, modules, etc.) are *suspects* that may be responsible for the failure [1]–[3]. Whether using simulation-based or formal-based techniques [1], [4], [5], a debugger returns a list of suspects (*i.e.,* locations) where a fix can be applied either by an engineer or an automated process to rectify the failure [6].

Although many advancements have been made recently in the field of design debugging [1]–[3], current techniques must scale to larger problems to be adopted by the industry. For example, most modern debuggers operate on sequential problems by constructing an Iterative Logic Array (ILA) or a time frame expansion model [4], [7]. In this representation, the combinational portion (*i.e.,* transition relation) of the circuit is replicated as many times as there are clock cycles in the error trace or counter-example. Thus, with large designs and long error traces, the ILA model can lead to overwhelming memory requirements and performance degradation.

This problem is exacerbated when dealing with long error traces from simulation-based (*i.e.* dynamic) verification. Unlike counter-examples from equivalence checking or property checking tools, simulation-based techniques, which account for over 90% of verification performed [8], are especially challenging as trace lengths can easily exceed thousands of clock cycles. As a consequence, dealing with long error traces is one of the premiere challenges of automated debugging today.

[01]Vennsa Technologies, Inc., Toronto, ON M5V 3B1 (sean@vennsa.com)
[02]University of Toronto, ECE Department, Toronto, ON M5S 3G4 (veneris@eecg.toronto.edu)

In the quest for scalable automated debugging tools, this paper introduces *Bounded Model Debugging (BMD)*. BMD is not a stand-alone debugger, but is a systematic methodology that can help existing debuggers cope effectively with long traces. At its core, BMD is motivated by the observation that errors are often excited and observed within close temporal proximity. In other words, there is a high likelihood that the cause of a failure is relatively close to the failure point. This observation is exploited manually in practice by verification engineers coping with the long traces when devising a divide and conquer approach.

The BMD methodology begins by constructing a small debugging problem based on a subsequence of the error trace containing the first observed failure and some prior clock cycles. The problem can be solved by existing debugging algorithms, and analysis of the solutions determines whether preceding clock cycles must be examined to guarantee completeness of the solutions. If required, a slightly larger problem is constructed as the process re-iterates. In this manner, BMD tackles the computationally intensive debugging problems by formulating incrementally larger ones in succession as needed. Furthermore, if resources are exhausted and complete debugging is not feasible, a subset of the solutions can be returned to the engineer. This work develops theory and performance enhancing techniques to demonstrate the correctness and efficiency of the methodology.

Experiments validate the motivation and analysis presented through a large set of problems using `OpenCores` [9] and real-life industrial designs. The BMD technique exhibits a run-time improvement of as much as two orders of magnitude when compared to a conventional debugging methodology. Furthermore, it is more robust as it solves more than 93% of the problems when compared to just 35% using a stand-alone debugger.

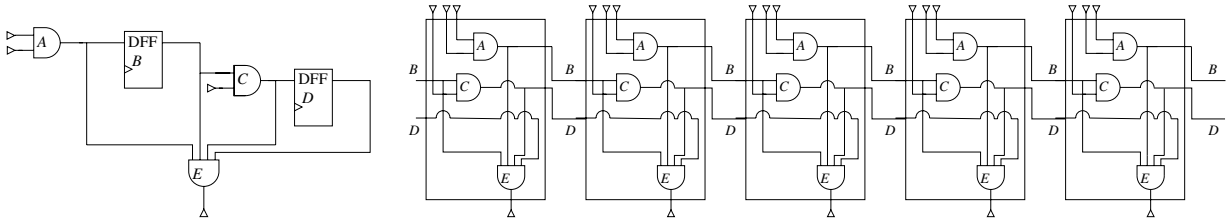
The remaining paper is organized as follows. Section II motivates this work through an illustration and probabilistic analysis. Section III sets the groundwork for the basic methodology while Section IV presents performance improvements. Section V contains the experiments and the last section concludes the paper.

II. Motivation

A. Illustrative Example

In combinational circuits, because there are no memory elements, errors are excited in the same clock cycle that the failing behavior is observed. In sequential circuits, the situation can be much more complex since the erroneous behavior may propagate across many consecutive clock cycles as values get latched in memory elements until error effects are observed at a primary output. Hence, when debugging simulation traces in sequential machines, many clock cycles must be considered prior to the observation of the failure.

978-1-4244-5766-3/10/$26.00 ©2010 IEEE



(a) Sample pipeline circuit with single output (b) Five time frame ILA for circuit in (a)

Fig. 1. Motivating example for BMD

Consider the sequential circuit in Fig. 1(a). Here, assume that five clock cycles are necessary to observe the first error at the primary output. The ILA representation of five cycles shown in Fig. 1(b) is used to demonstrate how errors can be excited in different time frames to cause the observed failure without any knowledge of the input stimulus.

Considering gate A, notice that if the error is excited in the first two cycles, gate A cannot be the error source because there is no propagation path from A in cycle one or two to the primary output in cycle five. If it is the case that an error on gate A is excited in cycle three, this failure cannot be observed in time frames three or four since the failure is first observed in time frame five. Similarly, the error may be excited in time frame four, but a failure cannot be observed in that time frame. Finally, the error can be both excited and observed in time frame five. This informal analysis, without knowledge of input stimulus, provides us with the intuition that the likelihood of an error source being present in a clock cycle increases as we approach the cycle where the failure is observed. This observation is analyzed probabilistically next.

B. Probabilistic Analysis of Error Behavior

BMD is heuristic motivated by the empirical observation that functional errors are usually excited in temporal proximity to observation points such as primary outputs. The purpose of Proposition 1, below, is to probabilistically explain the intuition developed in the previous section. Note that in order to simplify the proof, the proposition contains assumptions that may not be exact in practice and thus the result cannot be generalized. However, the result can provide insight into the effectiveness BMD and its empirical findings of Section V.

Proposition 1: Assuming that a single error is excited in clock cycle 1 and no other errors are excited in any other clock cycles, let $prop_i$ be the probability of the error propagating from cycle i to $i+1$ and obs_i the probability of observing a failure in clock cycle i, given that the error has propagated to that cycle. Also assume that the input vector sequences are temporally independent and stationary random sequences. Then, the probability of observing the first failure in clock cycle d is $p_d = \prod_{i=1}^{d-1} prop_i \times \prod_{i=1}^{d-1} (1-obs_i) \times obs_d$.

Proof: Let $W_i = \{$an error propagates from cycle i to cycle $i+1$ if it has propagated to cycle i $\}$, and $O_i = \{$a failure is observable in cycle i if an error has propagated to cycle i $\}$, and $E_1 = \{$an error is excited in clock cycle 1$\}$. Probability p_d can be stated in terms of events W_i, O_i, and E_1:

$p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \cap \bigcap_{i=1}^{d-1} \overline{O}_i \cap O_d \;\middle|\; E_1\right)$. By applying the identity $\mathcal{P}(A \cap B \mid C) = \mathcal{P}(A \mid C) \times \mathcal{P}(B \mid A \cap C)$, we get $p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) \times \mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} \overline{O}_i \cap \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$. Here, the events O_d and $\bigcap_{i=1}^{d-1} \overline{O}_i$ are conditionally independent of $E_1 \cap \bigcap_{i=1}^{d-1} W_i$. Thus, $\mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} \overline{O}_i \cap \bigcap_{i=1}^{d-1} W_i \cap E_1\right) = \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$. As a result, p_d can be simplified

$p_d = \mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) \times \mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$.

One of the assumptions made is that input vectors in successive cycles are all (temporally) independent. Thus, any W_i is independent of W_j for all cycles $i \neq j$: $\mathcal{P}(W_i \cap W_j \mid E_1) = \mathcal{P}(W_i \mid E_1) \times \mathcal{P}(W_j \mid E_1)$.

As a result, $\mathcal{P}\left(\bigcap_{i=1}^{d-1} W_i \;\middle|\; E_1\right) = \prod_{i=1}^{d-1} \mathcal{P}(W_i \mid E_1)$.

Similarly, by the assumption, any O_i is independent of O_j for all cycles i and j:

$\mathcal{P}\left(O_i \cap O_j \;\middle|\; \bigcap_{k=1}^{d-1} W_k \cap E_1\right) = \mathcal{P}\left(O_i \mid \bigcap_{k=1}^{d-1} W_k \cap E_1\right) \times \mathcal{P}\left(O_j \mid \bigcap_{k=1}^{d-1} W_k \cap E_1\right)$.

As a result, $\mathcal{P}\left(\bigcap_{i=1}^{d-1} \overline{O}_i \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right) = \prod_{i=1}^{d-1} \mathcal{P}\left(\overline{O}_i \mid \bigcap_{k=1}^{d-1} W_k \cap E_1\right)$.

Using the above, p_d can be simplified to:

$p_d = \prod_{i=1}^{d-1} \mathcal{P}(W_i \mid E_1) \times \prod_{i=1}^{d-1} \mathcal{P}\left(\overline{O}_i \mid \bigcap_{k=1}^{d-1} W_k \cap E_1\right) \times \mathcal{P}\left(O_d \;\middle|\; \bigcap_{i=1}^{d-1} W_i \cap E_1\right)$.

In the assumptions, $prop_j$ and obs_j are defined as:

$prop_j = \mathcal{P}(W_j \mid E_1)$ and $obs_j = \mathcal{P}\left(O_j \;\middle|\; \bigcap_{i=1}^{j-1} W_i \cap E_1\right)$ for some cycle j.

Using these definitions, p_d can be presented as

$p_d = \prod_{i=1}^{d-1} prop_i \times \prod_{i=1}^{d-1} (1-obs_i) \times obs_d$

■

We can simplify p_d by assuming that $prop_i = prop$ and $obs_i = obs$ that remain constant for all cycles i resulting in $p_d = prop^{d-1} \times (1-obs)^{d-1} \times obs$. This simplified relationship is plotted in Figure 2 for three values of $prop = obs = \{0.1, 0.5, 0.9\}$. For values at $d = 1$ we have $p_d = P(O_1|E_1) = obs$. The negative exponential relationship is clear as the three curves are no longer visible when $d > 6$. Although overly simplified, the expression for p_d aligns with the observations made in the field as well as the experimental results of Section V.

III. Bounded Model Debugging

The Bounded Model Debugging (BMD) methodology proposed in this paper leverages the insight that errors are more likely to be closer rather than farther from the failure observation point. Note that we do not propose a debugger but a complete and systematic technique that can be used with existing debuggers. BMD allows debugging techniques to

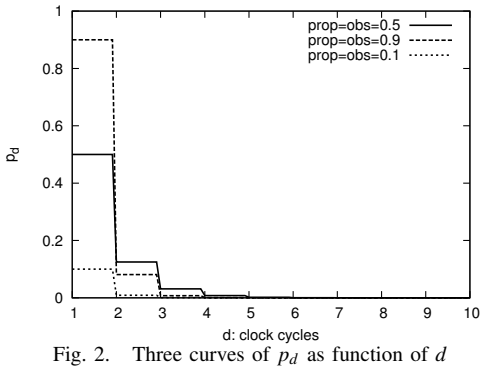


Fig. 2. Three curves of p_d as function of d

find error sources called *suspects* by considering only a subset of the error trace. Conceptual analogies can be drawn with Bounded Model Checking as both techniques incrementally operate on limited models of the problem to efficiently manage the available resources [7], [10].

We define a *suffix* as a subsequence of an error trace that includes the last clock cycle where the failure is first observed. Given an error trace of k_f clock cycles, the BMD methodology starts by considering a short suffix ranging from clock cycle k_1 to k_f, where k_1 is a cycle greater than one but less than k_f. In the remaining of this paper, v_{BMD} refers to the suffix of the error trace. Note that the suffix not only contains the input vector sequence but also the expected output vector sequence and the set of state values for state elements in clock cycle k_1-1. These values can be captured by simulating the circuit from clock cycle 1 to k_1-1 under the input stimulus sequence. Using the suffix v_{BMD}, a conventional debugger [1] will solve for the error suspects using a smaller ILA of size k_f-k_1 rather than size k_f.

Due to the smaller ILA size, the above procedure can provide results faster while requiring fewer memory resources. However, the solution set may be incomplete as some error sources may be excited in clock cycles prior k_1. In this case a longer suffix starting from cycle $k_2 < k_1$ is required to ensure completeness. The process continues with i BMD iterations with a suffix starting from cycle k_i until all solutions are found.

We can detect whether all solutions are found in any iteration i by asking the debugger if any memory elements (*i.e.* flip-flops or latches) in cycle k_i are found as solutions. Since these solutions point to the initial state of the debugging problem, we call these *initial state suspects*. When a debugger finds any initial state suspects, it indicates that an error may be excited in cycles prior to k_i (since state elements are the only components that can propagate signal values across clock cycles) and a longer suffix must be analyzed.

Figure 3 helps illustrate the BMD formulation as it presents an ILA representation for a trace of length k_f clock cycles. Each time frame is labelled and corresponds to the unrolling of the transition relation in the given clock cycle with inputs and outputs shown on the top and bottom of the time frames

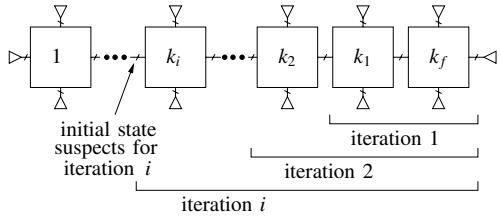


Fig. 3. Illustration of BMD formulation for multiple iterations

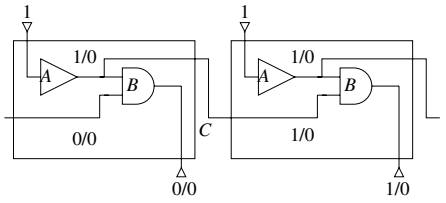


Fig. 4. Two clock cycle example annotated with `correct/erroneous` values

respectively. For each BMD iteration, a label shows the subsequence of time frames used. In iteration i, the initial state suspects are shown as the current states in cycle k_i.

A. Impact on Error Cardinality

For most automated debuggers, a parameter *maxN* is defined by the user to indicate the maximum number of suspects sought [1]. Subsequently, for performance enhancement reasons, the engine attempts to find N error locations while sequentially increasing the value of $N = 1,2,\ldots,maxN$. The proposed BMD methodology can impact the error cardinality *maxN* used by automated debuggers as follows.

Consider the two-cycle ILA in Figure 4 where the error is on gate `A`. In this case, when employing BMD with an initial suffix of length one and looking for $N = 1$ errors, only suspect gate `B` is found as a solution. More specifically, only the function of gate `B` can be changed to rectify the error observed at the primary output. The erroneous gate `A` and initial state suspect `C` are not returned as solutions since neither one can fix the failure on its own in the second cycle. As a result, because `C` is not contained in the solution set, the suffix length will not be increased and the method terminates erroneously.

This erroneous behavior is due to the fact that the error from gate `A` in the first cycle propagates to two distinct elements in the second cycle (`C` and `B`), whose combined effect result in the observed error. Thus the debugging problem requires a cardinality $N = 2$ with suffix length of one. For example, if $N = 2$ with $k_1 = 2$, then the solution {`B`, `C`} is returned. Since `C` is also an initial state suspect, the suffix length will be increased and the algorithm will iterate successfully.

The above example shows that the maximum error cardinality for BMD may be different than *maxN* set by the user. The following theorem presents an upper bound for the error cardinality mandated to find all initial state suspects and guarantee completeness. This estimate is refined in Section IV.

Theorem 1: Consider an erroneous circuit with *maxN* errors and a trace v where some errors are excited prior to clock cycle k_i. The BMD methodology guarantees to debug cycles prior to k_i if the maximum error cardinality is $maxN_{BMD} = N_{DFF} + maxN$, where N_{DFF} is the total number of state elements.

Proof: For any debugging problem where the first failure is observed in cycle k_f, consider the case where *maxN* errors are excited *both* before and after some clock cycle k_i. In the worst case the error effects are latched in all state elements in clock cycle k_i. If BMD is applied using a trace v_{BMD} of length k_f-k_i, then error suspects must be found on every *maxN* gate as well as every state element. Thus in order to allow BMD to debug prior cycle to k_i, a maximum error cardinality of $maxN_{BMD} = N_{DFF} + maxN$ must be used. ■

Under the suffixes of different BMD iterations, the error cardinality can increase, as shown above, or decrease. At every iteration the value of N must be reset to 1 regardless of its value in previous iterations to ensure that the smallest cardinality solutions are found.

IV. Performance Enhancements

The previous section introduced the basic BMD methodology while guaranteeing solution completeness. This section presents several performance enhancing techniques.

A. Reducing the Number of Initial Error Suspects

One improvement relates to the set of initial state suspects. As stated by Theorem 1, the maximum error cardinality for a BMD problem can grow according to the number N_{DFF} of state elements in the circuit. Since the complexity of the debugging problem grows exponentially with the error cardinality [1] it becomes important to develop techniques to reduce the number of initial error suspects.

One way to avoid a large increase in the error cardinality, is to group all initial suspects together as a single suspect. Since any solution set with initial state suspects requires increasing the length of the suffix for future iterations of BMD, there is no need to distinguish which initial state suspects are found. This is formalized in the theorem that follows.

Theorem 2: Consider an erroneous circuit with *maxN* errors and a trace v where some errors are excited before clock cycle k_i. The BMD methodology guarantees to debug cycles prior to k_i if the maximum error cardinality is $maxN_{BMD} = maxN + 1$ and all initial state suspects are grouped together.

Proof: For any debugging problem where the first failure is observed in cycle k_f, consider the case where *maxN* errors are excited *both* before and after some clock cycle k_i. In the worst case the error effects are latched in all state elements in clock cycle k_i. If BMD is applied using a trace v_{BMD} of length $k_f - k_i$, then error suspects must be found on every *maxN* gate as well as every state element. Since all state elements in cycle k_i are grouped together, for every initial state suspect sought the single group will be found. Thus in order for BMD to debug prior cycle to k_i, a maximum error cardinality of $maxN_{BMD} = maxN + 1$ must be used. ■

B. Reusing solutions

Another improvement relates to the iterative nature of the BMD methodology. At every iteration, debugging problems with longer suffixes may contain solutions that are already found through previous iterations with smaller suffixes. For example, consider two suffixes, one from clock cycle k_i to k_f, while the other is from clock cycle k_j to k_f, where $k_i > k_j$. Every solution set s for the shorter suffix that does not contain any initial state suspects is also a solution for the longer suffix. In other words, since the interval k_j to k_f contains k_i to k_f, solution set s will also be a solution in the larger suffix.

This observation allows BMD to return viable solutions to the end user prior to completing the iterative process. Furthermore solutions found in previous iterations can be skipped to improve overall performance. In a SAT-based debugging framework for instance, this can be achieved by adding a conflict clause to the CNF [1] to block solutions from being found in subsequent iterations.

C. Overall Algorithm

Pseudo-code for the BMD methodology described in this paper, including the performance improvements of the previous section, is shown in Algorithm 1.

Initially, BMD uses the suffix from clock cycle $k_f - incr$ to clock cycle k_f as shown on line 3. The `while` loop shown from line 4 to line 27 comprises the BMD iterations where successive debugging problems are constructed with longer suffixes. On line 5 the initial state constraints are captured by simulating the circuit C for $k-1$ cycles, while on line 6, the stimulus, response and initial state values are combined to construct v_{BMD}. Grouping the initial state suspects as presented in Section IV-A and adding all the potential suspects to S is performed on line 7. On line 8, a debugger is called to solve the constructed problem with error cardinality N.

Algorithm 1 The complete BMD algorithm

```
1:  exit_condition = 0, N = 1
2:  Final_Solutions = ∅
3:  k = k_f − incr
4:  while (!exit_condition) do
5:      initial_states = get_current_states(C, k − 1)
6:      v_BMD = {initial_states, stimulus_{k→k_f}, response_{k→k_f}}
7:      S = suspect_locations ∪ group(initial_state_suspect)
8:      Solutions = debug(C, v_BMD, N, S)
9:      for all Solution ∈ Solutions do
10:         valid_solution = 1
11:         for all Suspects ∈ Solution do
12:             if (is_initial_state(Suspect)) then
13:                 k = k − incr
14:                 N = 0
15:                 valid_solution = 0
16:             end if
17:         end for
18:         if (valid_solution == 1) then
19:             Final_Solutions = Final_Solutions ∪ Solution
20:         end if
21:     end for
22:     if (N == maxN + 1) then
23:         exit_condition = 1
24:     else
25:         N = N + 1
26:     end if
27: end while
28: return Final_Solutions
```

Once solutions are found by the debugger, determining to extend the length of the suffix is decided on line 12 based on whether the grouped initial state suspect is found. Lines 13–14 increase the length of the suffix and reset the error cardinality. When a solution does not contain the initial state suspect, the solutions are added to the final set as shown on line 19. Finally, the BMD process terminates when the maximum user defined cardinality *maxN* is reached in line 22. Not shown here, are termination conditions based on resource limits such as time-out and memory-out.

V. Experiments

In this section, we present experimental results of the proposed BMD methodology. All experiments are conducted on a single core of a Core 2 Quad 2.66GHz machine with 8GB of memory. The debugger used is a hierarchical sequential engine developed in C++ based on the concepts of [11] with a Verilog frontend to allow for RTL-based debugging. The SAT solver used is MiniSAT [12]. In the following this tool is referred to as the *stand-alone debugger*.

The circuits selected for experiments are Verilog RTL designs from OpenCores [9] as well as three industrial designs (fxu, rx_comm, s_comm) provided to the authors by semiconductor firms. In each of these designs one or more errors are added at the RTL level. For example these errors may be wrong state transitions, incorrect RTL operations, or even wrong module instantiations. It is important to emphasize that these errors at the RTL often translate into dozens of

Problem	problem stats			stand-alone debugger			proposed BMD				
	# gates	# DFFs	# cyc (k_f)	run-time (s)	# sols	found	run-time (s)	# iters	# sols	iter found	improv. (×)
ac97_ctrl-1	25310	2346	978	2613.62	49	yes	204.57	10	7	0	6.09
ac97_ctrl-2	25288	2345	670	1245.19	34	yes	747.24	10	13	1	1.67
div64bits-1	74846	5512	108	713.01	21	yes	1264.49	10	20	2	0.56
fdct-1	377801	5717	182	**MO**	N/A	no	**TO**	5	38	0	N/A
fdct-2	377801	5717	186	**MO**	N/A	no	**TO**	4	48	2	N/A
fpu-1	82371	1083	316	2108.97	6	yes	201.01	4	6	1	10.49
fpu-2	22953	515	640	**TO**	10	no	333.00	10	24	1	10.81
fxu-1	602673	29080	28	1958.15	32	yes	479.14	1	32	1	7.51
fxu-2	267423	12016	154	**TO**	3	no	174.36	1	28	1	4.09
mem_ctrl-1	46168	1145	681	2190.29	5	yes	22.43	1	5	1	97.65
mem_ctrl-2	46168	1145	757	**TO**	5	no	28.35	1	11	1	126.98
rx_comm-1	585641	30339	675	**MO**	N/A	no	452.97	1	30	1	7.95
rx_comm-2	585641	30339	253	**MO**	N/A	no	331.19	1	18	1	10.87
rx_comm-3	585632	30339	573	**MO**	N/A	no	369.09	1	5	1	9.75
rx_comm-4	220456	18333	180	2240.73	85	yes	**TO**	3	81	7	0.62
rx_comm-5	585265	30339	99	**TO**	54	no	275.79	1	15	1	13.05
rx_comm-6	585641	30339	560	**MO**	N/A	no	393.01	1	17	1	9.16
s_comm-1	779607	29967	212	**MO**	N/A	no	**TO**	4	21	1	N/A
s_comm-2	779607	29967	212	**MO**	N/A	no	**TO**	4	20	3	N/A
s_comm-3	779575	29967	212	**MO**	N/A	no	**TO**	4	14	1	N/A
s_comm-4	779607	29967	132	**MO**	N/A	no	**TO**	3	71	1	N/A
s_comm-5	790407	29967	132	**MO**	N/A	no	**TO**	3	39	2	N/A
spi-1	2942	185	251	973.18	65	yes	151.07	10	63	1	3.53
spi-2	2954	185	648	**MO**	N/A	no	106.47	10	57	1	33.81
vga-1	153837	17102	863	**MO**	N/A	no	553.35	3	63	1	6.51
vga-2	153837	17102	902	**MO**	N/A	no	1336.67	3	33	1	2.69
vga-3	155370	17206	175	1626.64	63	yes	685.95	3	83	1	2.37
vga-4	154137	17138	209	1531.70	33	yes	163.03	1	33	1	9.40
vga-5	154609	17146	381	**MO**	N/A	no	2982.43	5	29	3	1.21
vga-6	153837	17102	849	**MO**	N/A	no	166.52	1	8	1	21.62
wb-1	4479	251	269	466.03	14	yes	553.35	3	63	1	0.84

TABLE I
CIRCUIT AND PERFORMANCE STATISTICS WITHOUT BMD

error locations at the gate-level. Every instance of the designs with an inserted error is a debugging problem used in the experiments. Each debugging problem has a corresponding error trace which includes stimulus vectors and expected response vectors provided by the testbench.

The experimental results presented in Table I are grouped in three sections. Section one provides a summary of the debugging problems. Section two, shows the performance and results of the stand-alone debugger. While section three summarizes the results of the proposed BMD methodology. In section one of Table I, columns one, two and three label the debugging problem, and show the gate and DFF count, respectively. Column four shows the number of clock cycles in the entire error trace, corresponding to the first clock cycle k_f where a failure is observed. The problems used are specifically chosen because of their large circuit size (over 100K gates), long error trace (hundreds of clock cycles) or both. This combination results in hard problems that push the capabilities of the debuggers.

The next three columns of Table I present debugging statistics when using the stand-alone debugger. Column five shows the run-time in seconds required to solve each problem. Column six presents the number of solutions found, equivalent to the total number of equivalent error locations found with $maxN = 1$. Column seven states whether the actual inserted RTL error is found as one of the solutions. In cases where more than one hour of CPU is used, a time-out (**TO**) is declared and where more than 8GB of memory is required, a memory-out (**MO**) is declared. Note that some solutions may be available for time-out cases, whereas no solutions are found when a debugging formulation uses excessive memory. In summary, of the 31 debugging problems, three time-out, 17 memory-out, and the inserted error is found in only 11 or 35% of all cases.

The BMD methodology introduced in this paper is implemented according to Algorithm 1. An initial suffix length of 10 clock cycles is used as well as an increment of 10 clock cycles each time the suffix is increased. A maximum limit of 100 clock cycle is set as a hard limit, where the BMD methodology terminates. The performance of BMD is presented in the last five columns of Table I. In this section, column one presents the run-time in seconds required by BMD to solve each problem. Column two shows the number of debugging iterations performed until the process terminates. The corresponding total number of solutions found by all iterations are shown in column three. When the inserted error is found, the iteration in which the error is found is listed in column four. If the inserted error is not found, a zero (0) is listed in the column. The final column presents the performance improvement achieved by the proposed methodology over the stand-alone debugger.

The benefit of the BMD methodology is apparent based on multiple criteria. First notice that none of the problems solved with our methodology exceed the 8GB memory limit while 17 instance resulted in a memory-out with the stand-alone debugger. Instead, with BMD, eight problems run over the one hour time limit. It is clear that our technique provides a trade-off between the time and memory resources. This trade-off is

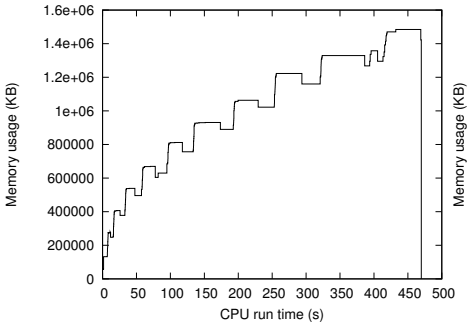


Fig. 5. fpu-2: memory vs. run-time

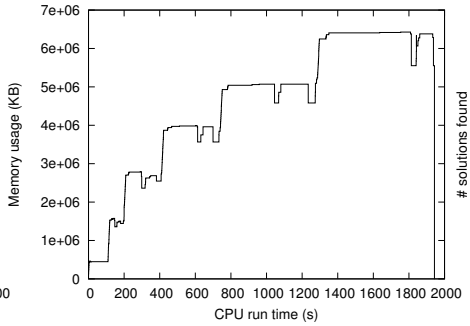


Fig. 6. vga-1: memory vs. run-time

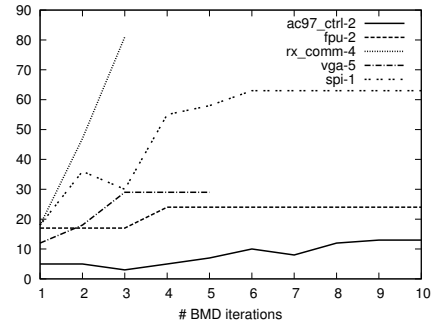


Fig. 7. # solutions vs. BMD iterations

seen favorably because the overall number of problems where the inserted error is found increases from 11 to 29 when using BMD. In practice, the complete problem need not be solved in order to find the error source or to provide vital debugging information to the user.

When using BMD, as shown in the second to last column, for only two problems the inserted RTL error is not found versus 20 with the stand-alone debugger. These two cases are `ac97_ctrl-1` where the maximum suffix length of 100 clock cycles is reached and `fdct-1` where the time-out limit of one hour is reached. Furthermore, notice that for all problems our approach finds at least some solutions versus 17 problems for which the stand-alone debugger did not find any solutions due to memory-outs. Again, this data favors the memory versus time trade-off of our technique.

The data in Table I reaffirms the probabilistic analysis of Section II-B that errors are excited in temporal proximity to the failure point. In the column *# iter*, 11 of 31 problems only require one BMD iteration or a suffix of 10 cycles to debug the problem completely. On average less than 15% of the original trace length is used. Without considering cases that time-out, only 6 of 23 problems or 26% of cases require more than 100 clock cycles to provide complete solutions.

Finally, observe the run-time improvement of the our methodology over the stand-alone debugger shown in the last column of Table I. Here improvements are achieved from 1.21× to 126.98×, or two orders of magnitude. Only in three cases, `div64bits-1`, `rx-comm4` and `wb-1` a performance degradation is observed because the multiple iterations result in longer run-time than running the stand-alone debugger. However, it is clear that BMD is very effective for the vast majority of problems.

Since our technique only uses as much memory as required by the suffix, memory requirements are only as much as the stand-alone debugger when the entire trace must be analyzed. Figure 5 and 6 provide more insight as they plot the memory requirement as a function of CPU time for problems `fpu-2` and `vga-5`. The memory requirement graph follows a rising step pattern each time the suffix length is increased. For example, in Figure 6, there are five distinct plateaus corresponding to the debugger solving problems with suffixes of length 10, 20, 30, 40 and 50. Notice that at each iteration, the solve time appears to increase at a faster rate than the suffix length. For example, the first iteration, which requires approximately 1.5 GB, takes under 100 seconds to solve, while the last iteration, which requires approximately 6.5 GB, takes approximately 600 seconds to solve.

The final analysis of the BMD methodology is with respect to the number of solutions found as a function of iterations. As shown in Figure 7, for the sample problems selected, the number of solutions found by BMD increases initially and plateaus in later iterations. Notice that the number of solutions does not always increase, since some solutions which may contain initial state suspects in prior iterations may be removed as solutions in future iterations. This graph portrays our methodology favorably as it indicates that increasing the suffix length after a certain point does not result in any more new solutions. As a result, the BMD approach of starting with a small suffix and systematically increasing the suffix length appears to be effective for debugging.

VI. Conclusion

This work introduces the bounded model debugging methodology to efficiently and systematically tackle problems with long error traces. The contribution is based on the empirical observation that errors are excited and failures are observed in temporal proximity. The methodology proposed is found to be faster than a conventional debugger in 90% of cases. Furthermore it is more robust, as the error is found in over 93% of problems compared to 35% without BMD. Overall, the proposed technique allows large problems with very long traces to be handled in an efficiency manner by existing debuggers.

References

[1] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[2] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Design, Automation and Test in Europe*, 2007, pp. 1182–1187.

[3] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

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

[5] A. Jain, V. Boppana, R. Mukherjee, J. Jain, M. Fujita, and M. Hsiao, "Testing, verification, and diagnosis in the presence of unknowns," in *VLSI Test Symp.*, 2000, pp. 263–269.

[6] K.-H. Chang, I. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," *IEEE Trans. on Comp.*, p. to appear, 2008.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[8] International Techonology Roadmap for Semiconductors, "ITRS 2006 Update," 2008, `http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm`.

[9] OpenCores.org, 2008, `http://www.opencores.org`.

[10] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of sat-based model checking techniques in an industrial environment." in *CHARME*, 2005, pp. 254–268.

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

[12] N. S. N. Een, "An Extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 333–336.